Version 16

Scripting Guide

"The real voyage of discovery consists not in seeking new landscapes, but in having new eyes."

Marcel Proust

16.0

Get the Most from JMP

Whether you are a first-time or a long-time user, there is always something to learn about JMP.

Visit JMP.com to find the following:

- live and recorded webcasts about how to get started with JMP
- video demos and webcasts of new features and advanced techniques
- details on registering for JMP training
- schedules for seminars being held in your area
- success stories showing how others use JMP
- the JMP user community, resources for users including examples of add-ins and scripts, a forum, blogs, conference information, and so on

https://www.jmp.com/getstarted

Contents

Scripting Guide

12 **Scripting Graphs** 631
Editing and Creating Two-Dimensional Graphs

13 **Three-Dimensional Scenes** 723
Scripting in Three Dimensions

Chapter **1**

Learn about JMP

Documentation and Additional Resources

Learn about JMP documentation, such as book conventions, descriptions of each JMP document, the Help system, and where to find additional support.

Contents

Formatting Conventions in JMP Documentation

These conventions help you relate written material to information that you see on your screen:

- Sample data table names, column names, pathnames, filenames, file extensions, and folders appear in Helvetica (or sans-serif online) font.

- Code appears in `Lucida Sans Typewriter` (or monospace online) font.

- Code output appears in *Lucida Sans Typewriter* italic (or monospace italic online) font and is indented farther than the preceding code.

- **Helvetica bold** formatting (or bold sans-serif online) indicates items that you select to complete a task:

    – buttons

    – check boxes

    – commands

    – list names that are selectable

    – menus

    – options

    – tab names

    – text boxes

- The following items appear in italics:

    – words or phrases that are important or have definitions specific to JMP

    – book titles

    – variables

- Features that are for JMP Pro only are noted with the JMP Pro icon **JMP PRO** . For an overview of JMP Pro features, visit https://www.jmp.com/software/pro.

Note: Special information and limitations appear within a Note.

Tip: Helpful information appears within a Tip.

JMP Help

JMP Help in the Help menu enables you to search for information about JMP features, statistical methods, and the JMP Scripting Language (or *JSL*). You can open JMP Help in several ways:

- Search and view JMP Help on Windows by selecting **Help > JMP Help**.

- On Windows, press the F1 key to open the Help system in the default browser.

- Get help on a specific part of a data table or report window. Select the Help tool 💡 from the **Tools** menu and then click anywhere in a data table or report window to see the Help for that area.

- Within a JMP window, click the **Help** button.

Note: The JMP Help is available for users with Internet connections. Users without an Internet connection can search all books in a PDF file by selecting **Help > JMP Documentation Library**. See "JMP Documentation Library" on page 24 for more information.

JMP Documentation Library

The Help system content is also available in one PDF file called *JMP Documentation Library*. Select **Help > JMP Documentation Library** to open the file. If you prefer searching individual PDF files of each document in the JMP library, download the files from https://www.jmp.com/documentation.

The following table describes the purpose and content of each document in the JMP library.

Document Title	Document Purpose	Document Content
Discovering JMP	If you are not familiar with JMP, start here.	Introduces you to JMP and gets you started creating and analyzing data. Also learn how to share your results.
Using JMP	Learn about JMP data tables and how to perform basic operations.	Covers general JMP concepts and features that span across all of JMP, including importing data, modifying columns properties, sorting data, and connecting to SAS.

Document Title	Document Purpose	Document Content
Basic Analysis	Perform basic analysis using this document.	Describes these Analyze menu platforms: • Distribution • Fit Y by X • Tabulate • Text Explorer Covers how to perform bivariate, one-way ANOVA, and contingency analyses through Analyze > Fit Y by X. How to approximate sampling distributions using bootstrapping and how to perform parametric resampling with the Simulate platform are also included.
Essential Graphing	Find the ideal graph for your data.	Describes these Graph menu platforms: • Graph Builder • Scatterplot 3D • Contour Plot • Bubble Plot • Parallel Plot • Cell Plot • Scatterplot Matrix • Ternary Plot • Treemap • Chart • Overlay Plot The book also covers how to create background and custom maps.
Profilers	Learn how to use interactive profiling tools, which enable you to view cross-sections of any response surface.	Covers all profilers listed in the Graph menu. Analyzing noise factors is included along with running simulations using random inputs.

Document Title	Document Purpose	Document Content
Design of Experiments Guide	Learn how to design experiments and determine appropriate sample sizes.	Covers all topics in the DOE menu.
Fitting Linear Models	Learn about Fit Model platform and many of its personalities.	Describes these personalities, all available within the Analyze menu Fit Model platform: • Standard Least Squares • Stepwise • Generalized Regression • Mixed Model • MANOVA • Loglinear Variance • Nominal Logistic • Ordinal Logistic • Generalized Linear Model

Document Title	Document Purpose	Document Content
Predictive and Specialized Modeling	Learn about additional modeling techniques.	Describes these Analyze > Predictive Modeling menu platforms: • Neural • Partition • Bootstrap Forest • Boosted Tree • K Nearest Neighbors • Naive Bayes • Support Vector Machines • Model Comparison • Model Screening • Make Validation Column • Formula Depot Describes these Analyze > Specialized Modeling menu platforms: • Fit Curve • Nonlinear • Functional Data Explorer • Gaussian Process • Time Series • Matched Pairs Describes these Analyze > Screening menu platforms: • Modeling Utilities • Response Screening • Process Screening • Predictor Screening • Association Analysis • Process History Explorer

Document Title	Document Purpose	Document Content
Multivariate Methods	Read about techniques for analyzing several variables simultaneously.	Describes these Analyze > Multivariate Methods menu platforms: • Multivariate • Principal Components • Discriminant • Partial Least Squares • Multiple Correspondence Analysis • Structural Equation Models • Factor Analysis • Multidimensional Scaling • Item Analysis Describes these Analyze > Clustering menu platforms: • Hierarchical Cluster • K Means Cluster • Normal Mixtures • Latent Class Analysis • Cluster Variables
Quality and Process Methods	Read about tools for evaluating and improving processes.	Describes these Analyze > Quality and Process menu platforms: • Control Chart Builder and individual control charts • Measurement Systems Analysis • Variability / Attribute Gauge Charts • Process Capability • Model Driven Multivariate Control Chart • Legacy Control Charts • Pareto Plot • Diagram • Manage Spec Limits • OC Curves

Document Title	Document Purpose	Document Content
Reliability and Survival Methods	Learn to evaluate and improve reliability in a product or system and analyze survival data for people and products.	Describes these Analyze > Reliability and Survival menu platforms: • Life Distribution • Fit Life by X • Cumulative Damage • Recurrence Analysis • Degradation • Destructive Degradation • Reliability Forecast • Reliability Growth • Reliability Block Diagram • Repairable Systems Simulation • Survival • Fit Parametric Survival • Fit Proportional Hazards
Consumer Research	Learn about methods for studying consumer preferences and using that insight to create better products and services.	Describes these Analyze > Consumer Research menu platforms: • Categorical • Choice • MaxDiff • Uplift • Multiple Factor Analysis
Scripting Guide	Learn about taking advantage of the powerful JMP Scripting Language (JSL).	Covers a variety of topics, such as writing and debugging scripts, manipulating data tables, constructing display boxes, and creating JMP applications.
JSL Syntax Reference	Read about many JSL functions on functions and their arguments, and messages that you send to objects and display boxes.	Includes syntax, examples, and notes for JSL commands.

Additional Resources for Learning JMP

In addition to reading JMP help, you can also learn about JMP using the following resources:

- "JMP Tutorials"
- "Sample Data Tables"
- "Learn about Statistical and JSL Terms"
- "Learn JMP Tips and Tricks"
- "JMP Tooltips"
- "JMP User Community"
- "Free Online Statistical Thinking Course"
- "JMP New User Welcome Kit"
- "Statistics Knowledge Portal"
- "JMP Training"
- "JMP Books by Users"
- "The JMP Starter Window"

JMP Tutorials

You can access JMP tutorials by selecting **Help > Tutorials**. The first item on the **Tutorials** menu is **Tutorials Directory**. This opens a new window with all the tutorials grouped by category.

If you are not familiar with JMP, start with the **Beginners Tutorial**. It steps you through the JMP interface and explains the basics of using JMP.

The rest of the tutorials help you with specific aspects of JMP, such as designing an experiment and comparing a sample mean to a constant.

Sample Data Tables

All of the examples in the JMP documentation suite use sample data. Select **Help > Sample Data Library** to open the sample data directory.

To view an alphabetized list of sample data tables or view sample data within categories, select **Help > Sample Data**.

Sample data tables are installed in the following directory:

On Windows: C:\Program Files\SAS\JMP\16\Samples\Data

On macOS: \Library\Application Support\JMP\16\Samples\Data

In JMP Pro, sample data is installed in the JMPPRO (rather than JMP) directory.

To view examples using sample data, select **Help > Sample Data** and navigate to the Teaching Resources section. To learn more about the teaching resources, visit https://jmp.com/tools.

Learn about Statistical and JSL Terms

For help with statistical terms, select Help > Statistics Index. For help with JSL scripting and examples, select **Help > Scripting Index**.

Statistics Index   Provides definitions of statistical terms.

Scripting Index   Lets you search for information about JSL functions, objects, and display boxes. You can also edit and run sample scripts from the Scripting Index and get help on the commands.

Learn JMP Tips and Tricks

When you first start JMP, you see the Tip of the Day window. This window provides tips for using JMP.

To turn off the Tip of the Day, clear the **Show tips at startup** check box. To view it again, select **Help > Tip of the Day**. Or, you can turn it off using the Preferences window.

JMP Tooltips

JMP provides descriptive tooltips (or *hover labels*) when you hover over items, such as the following:

- Menu or toolbar options
- Labels in graphs
- Text results in the report window (move your cursor in a circle to reveal)
- Files or windows in the Home Window
- Code in the Script Editor

Tip: On Windows, you can hide tooltips in the JMP Preferences. Select **File > Preferences > General** and then deselect **Show menu tips**. This option is not available on macOS.

JMP User Community

The JMP User Community provides a range of options to help you learn more about JMP and connect with other JMP users. The learning library of one-page guides, tutorials, and demos is a good place to start. And you can continue your education by registering for a variety of JMP training courses.

Other resources include a discussion forum, sample data and script file exchange, webcasts, and social networking groups.

To access JMP resources on the website, select **Help > JMP User Community** or visit https://community.jmp.com.

Free Online Statistical Thinking Course

Learn practical statistical skills in this free online course on topics such as exploratory data analysis, quality methods, and correlation and regression. The course consists of short videos, demonstrations, exercises, and more. Visit https://www.jmp.com/statisticalthinking.

JMP New User Welcome Kit

The JMP New User Welcome Kit is designed to help you quickly get comfortable with the basics of JMP. You'll complete its thirty short demo videos and activities, build your confidence in using the software, and connect with the largest online community of JMP users in the world. Visit https://www.jmp.com/welcome.

Statistics Knowledge Portal

The Statistics Knowledge Portal combines concise statistical explanations with illuminating examples and graphics to help visitors establish a firm foundation upon which to build statistical skills. Visit https://www.jmp.com/skp.

JMP Training

SAS offers training on a variety of topics led by a seasoned team of JMP experts. Public courses, live web courses, and on-site courses are available. You might also choose the online e-learning subscription to learn at your convenience. Visit https://www.jmp.com/training.

JMP Books by Users

Additional books about using JMP that are written by JMP users are available on the JMP website. Visit https://www.jmp.com/books.

The JMP Starter Window

The JMP Starter window is a good place to begin if you are not familiar with JMP or data analysis. Options are categorized and described, and you launch them by clicking a button. The JMP Starter window covers many of the options found in the Analyze, Graph, Tables, and File menus. The window also lists JMP Pro features and platforms.

• To open the JMP Starter window, select **View** (**Window** on macOS) **> JMP Starter**.

• To display the JMP Starter automatically when you open JMP on Windows, select **File > Preferences > General**, and then select **JMP Starter** from the Initial JMP Window list. On macOS, select **JMP > Preferences > Initial JMP Starter Window**.

JMP Technical Support

JMP technical support is provided by statisticians and engineers educated in SAS and JMP, many of whom have graduate degrees in statistics or other technical disciplines.

Many technical support options are provided at https://www.jmp.com/support, including the technical support phone number.

Chapter **2**

Introduction

Welcome to the JMP Scripting Language

The JMP Scripting Language, or *JSL*, lets you write scripts to re-create results in JMP. Power users often develop scripts to extend JMP's functionality and automate a regularly scheduled analysis in production settings. If you do not want to learn JSL, JMP can write the scripts for you.

JSL is used to perform many actions:

- implements column formulas
- launches platforms
- interactively modifies platforms
- creates graphics

Contents

What JSL Can Do for You

JMP can automatically save scripts to reproduce any data table or analysis in its current state. You can pause anytime in your analysis to save a script to a script window (or *script editor*), in a data table, or in an analysis report. You can then modify the script as needed for future projects. When you are finished with your work, you can then save a script to reproduce your final results.

Here are some examples where JSL scripts can be helpful:

• Suppose you need to describe an analysis process in detail, from beginning to end. An example is to create an audit trail for a governing agency, or for peers reviewing your journal article.

• Suppose you have a set of analysis steps that should be followed routinely by your lab technicians.

• Suppose you fit the same model to new data every day, and the steps are always the same.

You can use JMP interactively as usual, save scripts to reproduce your work, and in the future run those scripts to reproduce your results.

There are a few things that JSL is not designed to do:

• JMP cannot record scripts while you are working. Though script-recording is a useful feature in some other scripting languages, it is less important for software like JMP, where the results are what matter. You cannot use script-recording to observe how a sequence of interactive steps is performed.

• JSL is not an alternative command-line interface for using the program.

Help with Learning JSL

There are several places within JMP to get help with writing or understanding a JSL script.

The Scripting Guide

The *Scripting Guide* begins with basic information (such as terminology and syntax) for JMP users who are not familiar with the scripting language. The *Scripting Guide* then progresses to more advanced information.

Chapters 2 through 4	Includes information about learning JSL, producing basic scripts, and introduces you to the JSL scripting environment.
Chapters 5 through 8	Introduces the building blocks of the language; working with basic data types, such as numbers and strings; writings lists, matrices, and associate arrays; namespaces; and the fundamentals of programming in JSL.
Chapters 9 through 13	Covers using JSL with objects in JMP, such as data tables, platforms, windows, and graphics.
Chapter 14	Describes how to write scripts that work with external programs, such as SAS, R, and Microsoft Excel.
Chapter 15	Describes how to organize files that you use in an analysis, perform analyses, and run scripts from one workspace.
Chapter 16	Introduces creating JMP applications in Application Builder, a drag-and-drop environment for visually designing windows with buttons, lists, graphs, and other objects. The chapter also describes how to use Add-In Builder to compile scripts into one easily shared file.
Chapter 17	Contains a collection of recipes, or script examples, that you can copy and modify for your own use.
Appendices A, B, C, and D	Provides information about compatibility issues with the previous version of JMP, provides tips for writing more efficient scripts, gives information about references, and defines JSL concepts and terminology.

The Scripting Index

The Scripting Index on the **Help** menu provides a brief description and the syntax for JSL functions, objects, and display boxes. Each entry includes an example that you can run and modify to test your own code. And an embedded log window lets you see messages as examples are run.

Notes:

- To display the Scripting Index entry for a function in a script, run the script, press Alt, and then double-click the function. You can also run the script, right-click the function, and select **Help Scripting Index.**

- To copy the syntax or description from the Scripting Index window, right-click the area that you want to copy and select **Copy Text**.

The Scripting Index window includes the following buttons:

✕   Click the **Clear** button to clear the search text box to begin a new search.

▼   Click the arrow to set search filter options and parameters.

🔄   After you edit the sample script in the Scripting Index, click this button to revert to the original script.

Note: When you edit an example script, the changes persist when you view other entries in the Scripting Index. To revert the script, click **Reset**, which appears above the script after you edit it.

Search Filter Options

Click the down arrow button next to the search box to refine your search.

Contains Terms   Returns items that contain a part of the search criteria. A search for "ease oom" returns messages such as "Release Zoom".

Contains Phrase   Returns items that contain the exact search criteria. A search for "text box" returns entries that contain "text" followed directly by "box" (for example, "Context Box" and "Text Box").

Starts With Phrase   Returns items that start with the search criteria.

Ends With Phrase   Returns items that end with the search criteria.

Whole Phrase   Returns items that consist of the entire string. A search for "text box" returns entries that contain only "text box".

Regular Expression   Enables you to use the wildcard (*) and period (.) in the search box. Searching for "get.*name" looks for items that contain "get" followed by one or more words. It returns "Get Color Theme Names", "Get Name Info", and "Get Effect Names", and so on.

Invert Result   Returns items that do not match the search criteria.

   Match All Terms   Returns items that contain both strings. A search for "t test" returns
      elements that contain either or both of the search strings: "Pat Test", "Shortest Edit Script"
      and "Paired t test".

   Ignore Case   Ignores the case in the search criteria.

   Match Whole Words   Returns items that contain each word in the string based on the Match
      All Terms setting. If you search for "data filter", and Match All Terms is selected, entries
      that contain both "data" and "filter" are returned.

   Search Category and Item Names   Returns category names (in the left column) and item
      names (in the middle column) that match the search criteria.

   Search Prototypes and Descriptions   Returns prototypes (the JSL syntax) and descriptions
      that match the search criteria.

   Search Examples   Returns examples that match the search criteria.

Let JMP Teach You JSL

The best JSL writer is JMP. You can work in JMP interactively and then save the results as a
script to reuse later. With simple modifications, your script can serve as a template for
speeding up routine tasks.

Because JSL is a very flexible language, you can reach your goals in many different ways. Here
is an example. Typically, the script that JMP saves for you specifies every detail of your
analysis, even if most of the details happen automatically by default. Does that mean that the
scripts that you write have to be just as complete and detailed? Not at all. You usually need to
specify only those details that you would select in the graphical user interface (GUI). For
example, if you open Big Class.jmp from the sample data folder and want to launch
Distribution for height, weight, and sex, the following script is all that is necessary:

```
Distribution( Y( :height, :weight, :sex ) );
```

Suppose you run the Distribution platform in the GUI and then select **Save Script > To Script
Window** from the red triangle menu for the report. The following script appears:

```
Distribution(
   Continuous Distribution( Column( :height ) ),
   Continuous Distribution( Column( :weight ) ),
   Nominal Distribution( Column( :sex ) ),
);
```

Both scripts give the same result.

Feel free to experiment with JSL. If you think something ought to be possible, it probably is.
Give it a try, and see what happens.

Terminology

Before you begin creating scripts, you should become familiar with basic JSL terms used throughout the *Scripting Guide*.

Operators and Functions

An *operator* is one- or two-character symbol (such as + or =) for features such as common arithmetic actions, scoping names, regular expressions, concatenating, and subscripting.

A *function* is a command that might contain additional information for the function to use.

Certain JSL functions work the same as operators but provide access to more complex actions. For example, the following two lines are equivalent:

```
2 + 3; // returns 5
Add( 2, 3 ); // returns 5
```

The first line uses the + operator. The second line uses the Add() function equivalent.

Although all JSL operators have function equivalents, not all functions have operator equivalents. For example, Sqrt(a) can be represented only by the Sqrt() function.

Note: In previous versions of JMP and its documentation, the terms *operators* and *functions* were used interchangeably. Now each term has a specific meaning.

Objects and Messages

An *object* is a dynamic entity in JMP, such as a data table, a data column, a platform results window, a graph, and so on. Most objects can receive messages that instruct the object to perform some action on itself.

A *message* is a JSL expression that is directed to an object. That object knows how to evaluate the message. In the following example, dt is the data table object. << indicates that a message follows. In the following example, the message tells JMP to create a summary table with the specified variables.

```
dt << Summary( Group( :age ), Mean( :height ) )
```

In this expression, dt is the name of a variable that contains a reference to a data table. You could use any name for this variable. The *Scripting Guide* commonly uses dt to represent data table references. Here are some of the more common names used to represent references to certain objects:

Abbreviation	Object
dt	data table
col	column in a data table
colname	the name of a column in a data table
obj	an object
db	display box

These variables are not pre-assigned references. Each one must be assigned prior to its use. In the following example, the global variable named A is assigned the value "Hello, World". When the Show(A) command is processed, the result is the value of A.

```
A = "Hello, World";
Show( A );
   A = "Hello, World";
```

Arguments and Parameters

An *argument* is additional information that you can provide to a function or message. For example, in Root(25), 25 is an argument to the Root() function. Root() acts on the argument that you provide and returns the result: 5.

Programming and scripting books commonly talk about parameters as well. A *parameter* is a description of the argument that a function accepts. For example, the general specification for Root() might be Root(number), where *number* is the parameter.

Parameter and *argument* express two perspectives of the same concept: information that a function needs.

For simplicity in the *Scripting Guide*, we use the word *argument* in both cases.

A *named argument* is an optional argument that you select from a predetermined set and explicitly define. For example, title("My Line Graph") in the Graph Box() function is a named argument because the title is explicitly defined as such.

```
Graph Box( title( "My Line Graph" ),
      Frame Size( 300, 500 ),
      Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
      Pen Color( "Blue" );
      Line( [10 30 70], [88 22 44] ));
```

Note that the Frame Size() arguments 300 and 500 are not named. The position of these arguments implies meaning; the first argument is always the width, the second argument is always the height.

Optional Arguments

Functions and messages require certain arguments, and other arguments are optional. You can include them, but you do not have to. In specifications, optional arguments are enclosed in angle brackets. For example:

```
Root( x, <n> )
```

The *x* argument is required. The *n* argument is optional.

Optional arguments often have a default value. For example, for `Root()`, the default value of *n* is 2:

Code	Output	Explanation
`Root( 25 )`	5	Returns the square root of 25.
`Root( 25, 2 )`	5	Returns the square root of 25.
`Root( 25, 3 )`	2.92401773821287	Returns the cube root of 25.

Or and the Vertical Bar Symbol

A single vertical bar (|) represents a logical `OR`. For brevity, | represents the word *or* when referring to alternative values.

For example, a pathname can be either absolute or relative. When you see an argument such as `absolute|relative`, this means that you enter *either* one of the following two options:

•   `absolute` indicates an absolute pathname.

•   `relative` indicates a relative pathname.

More than two options can also be strung together with a vertical bar in this way.

Script Formatting

Whitespace characters (such as spaces, tabs, and newlines) and capitalization are ignored in JSL. This means that the following two expressions are equivalent:

Expression 1:

```
sum = 0;
For( i = 1, i <= 10, i++,
    sum += i;
    Show( i, sum );
);
```

Expression 2:

```
Sum = 0;
For( i = 1, i <= 10, i++,
   Sum += i;
   Show( i, Sum );
);
```

You can format your script in any way that you like. However, the script editor can also format your script for you. The *Scripting Guide* uses the script editor's default formatting for capitalization, spaces, returns, tabs, and so on. See "Work with the Script Editor" on page 59 in the "Scripting Tools" chapter for more information about using the script editor.

Note: The only white space exception is two-character operators (such as <= or ++). The operators cannot be separated by a space.

Basic JSL Syntax

A JSL script is a series of expressions. Each expression is a section of JSL code that accomplishes a task. JSL expressions hold data, manipulate data, and send commands to objects.

Many expressions are nested message names. Message contents are enclosed in parentheses:

```
Message Name ( argument 1, argument 2, ... );
```

The meaning of JSL names depends on the context. The same name might mean one thing in a data table context and something entirely different in a function context. See "Rules for Name Resolution" on page 111 in the "JSL Building Blocks" chapter.

Almost anything that follows certain punctuation rules, such as matching parentheses, is a valid JSL expression. For example:

```
win = New Window( "Window Example",
   <<Modal,
   Text Box( "Hello, World" ),
   Text Box( "-----" ),
   Button Box( "OK" )
);
```

Notes:

- Names can have embedded spaces. See "Names" on page 101 in the "JSL Building Blocks" chapter.

- Message contents are enclosed in parentheses, which must be balanced. See "Parentheses" on page 98 in the "JSL Building Blocks" chapter.

- Items are separated by commas. See "Commas" on page 98 in the "JSL Building Blocks" chapter.

- JSL is not case sensitive; you can type "`text box();`" or "`Text Box()`".

- Messages are commonly nested inside other messages.

Book Conventions

In the *Scripting Guide*, the function names that are case sensitive are capitalized as you must use them. Arguments that are placeholders for actual choices are in `lowercase`. For example, `Connect Color` is a function that you need to type as is, and `color` stands for some color choice that you make yourself.

```
Connect Color(color);
```

In this case, the argument in parentheses must be some color value (for example, a JMP color number, or a supported color name like `"red"`, `"blue"`, and so on, or an RGB value given as a list, such as `{.75, .50, .50}`). Sometimes alternatives like these are shown with the vertical bar (|) character for "or," like this:

```
Connect Color( number | "color name" | {r,g,b} );
```

Syntax coloring is applied to scripts that you can paste into a script editor and then run.

Chapter **3**

Getting Started
Let JMP Write Your Scripts

You often have to produce the same reports for the same data on a regular basis. This chapter shows you how to let JMP write scripts for common tasks like importing text data, opening Microsoft Excel files, and producing reports. A final tutorial shows you how to put it all together into a single script to open an Microsoft Excel file and produce three reports automatically.

The *Scripting Guide* is written for users who are familiar with JMP but might not be familiar with JSL. For more information about performing common tasks, see *Using JMP*. *Discovering JMP* is also a good resource for learning basic concepts and understanding the JMP workflow.

Contents

Capturing a Script for an Analysis Report

Here are the basic steps for capturing a script to reproduce an analysis:

1. Launch a platform, such as Distribution.

2. Make any changes or additions that you need. For example, add tests and other graphs.

3. Capture the script to recreate your results.

You can save the script in the data table, so that if you send the data table to others, they can run your script and duplicate your reports.

Example

Follow these steps to produce a distribution report, capture the script to reproduce it, and save it to the data table.

Note: The data tables that you use in examples are located in JMP's Samples/Data folder.

1. Select **Help > Sample Data Library** and open Companies.jmp.

2. Select **Analyze > Distribution** to open the Distribution launch window.

3. Select Profits ($M) in the Select Columns box and click the **Y, Columns** button.

4. Click **OK**.

   The Distribution report window appears.

5. Click the Distributions red triangle and select **Stack** to make your report horizontal.

6. Click the red triangle next to Profits ($M) and deselect **Outlier Box Plot** to turn the option off.

7. Click the red triangle next to Profits ($M) and select **Test Mean**.

   The Test Mean window appears.

8. Type 500 in the **Specify Hypothesized Mean** box.

9. Click **OK**.

   The test for the mean is added to the report window.

   Now you have your customized report.

Figure 3.1 Customized Distribution Report
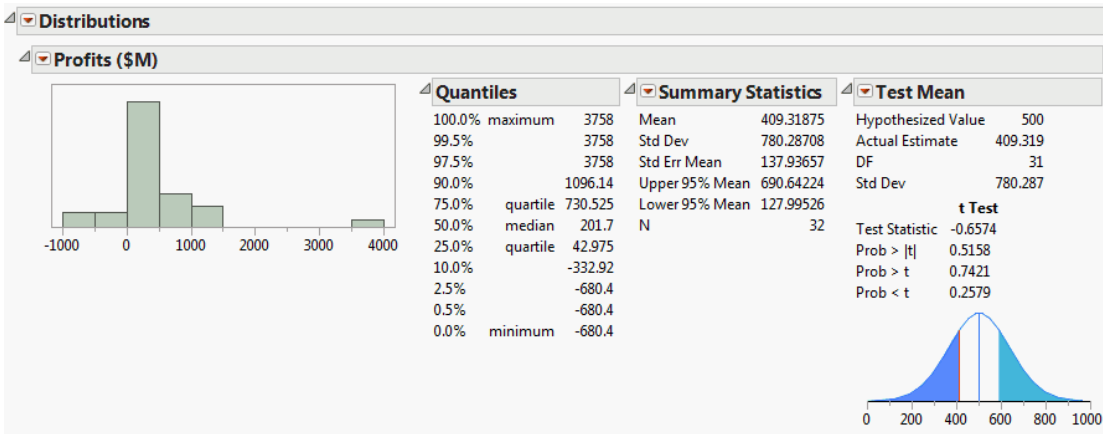
10. Click the Distributions red triangle and select **Save Script > To Data Table**.

Your data table now has a script named Distribution saved to it. Right-click the Distribution script and select **Edit** to see the script.

Figure 3.2 Distribution Script Saved to the Data Table
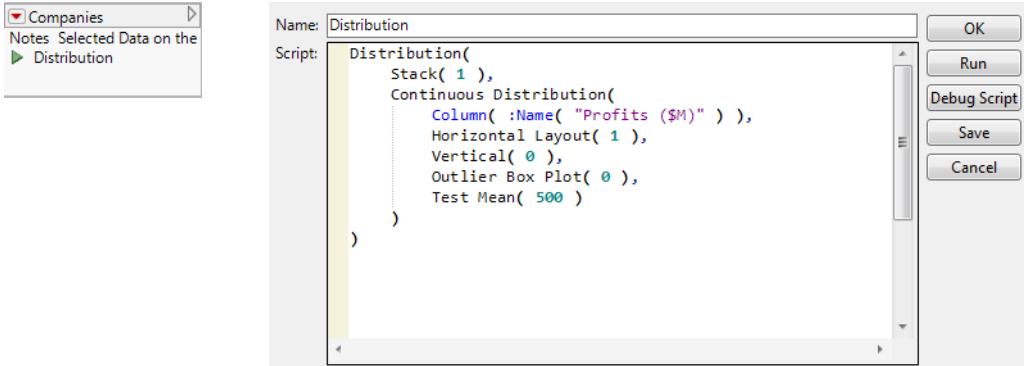
11. To run the script and reproduce your final report exactly, click the green triangle next to the script.

Capturing a Script for a Data Table

Here are the basic steps for capturing a script to reproduce a data table:

1. Open the data table.

2. Make any changes that you need. For example, add a script, correct values, add new columns.

3. Capture the script to recreate your data table.

Example

Use the data table from the previous example, where you saved a script to it.

1. In the data table, select the red triangle next to the data table's name.

2. Select **Copy Table Script**.

Figure 3.3 Copy the Table Script
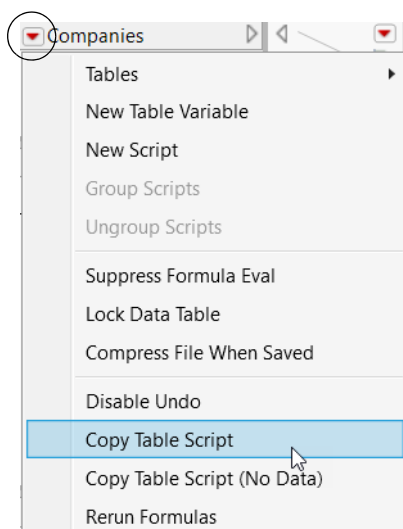
3. Open a script window by selecting **File > New > Script**.

4. Select **Edit > Paste**.

You now have a script that duplicates your data table. You can save this script and run it at any time to recreate your data table, with all its scripts attached.

Capturing a Script to Import a File

To capture a script that imports a file, you open the file in JMP. JMP automatically records the steps that occurred when you opened the file.

Import a Text File

1. Select **File > Open**.

The Open Data File window appears.

2. Select **Text Files** from the list next to **File name**.

3. In the Open as section, select **Data (Best Guess).**

   JMP formats the data based on tabs, commas, white space, and other characters in the text file.

4. Browse to select the file, and then select **Open**.

   The file is opened as a data table. The data table includes a script named Source. This JSL script imports your text file with the text import rules that you used.

5. Right-click the Source script and select **Edit**.

   You can copy this script, paste it into a new script window, and save it. Then you can run this script later to reimport the text file.

Tip: The import script is an Open() expression that specifies the text file and the import options to correctly import the file into JMP. The first part of this expression is the pathname to the specific file that you imported. If you save this script and want to run it a different place, you might need to edit the pathname so that it points to the text file. Pathnames are discussed in greater detail in "Path Variables" on page 145 in the "Types of Data" chapter.

Gluing Scripts Together

Suppose new data is saved out to an Microsoft Excel file once a week, and you need to produce the same reports every week. You could open the file and perform the same steps every week. However, creating a script that imports the new Microsoft Excel file into JMP and runs all analyses automatically is more efficient. The following example shows you how to set up your script and run it each week.

Import the Microsoft Excel File

1. Open a new script window (**File > New > Script**).

2. In your script window, enter the Open() expression to open the Solubil.xlsx sample import data file. The file is located in JMP's Samples/Import Data folder.

   ```
   dt = Open( "$SAMPLE_IMPORT_DATA/Solubil.xlsx" );
   ```

   Be sure to put the semicolon at the end of this expression, because you will add more expressions. The semicolon glues expressions together.

3. Run your script to import the Microsoft Excel file by selecting **Edit > Run Script**.

   The Microsoft Excel file opens as a data table.

Notes:

- You can also include the `Excel Wizard` argument in the `Open()` expression to preview the worksheet before importing it. See "Import Data from a Microsoft Excel File" on page 343 in the "Data Tables" chapter.

- You can specify an absolute or relative path to the file rather than using a path variable. For relative links, the script and file being opened must be in the same relative location each time you run the script. With absolute links, make sure that other users running the script have access to the file's location. See "Path Variables" on page 145 in the "Types of Data" chapter for more information about using pathnames.

Run Your Reports and Capture Their Scripts

You have three reports to produce: a distribution report, a 3D scatterplot, and a multivariate report. Perform each one using the GUI, and add its script to the script window.

1. With your new data table open, select **Analyze > Distribution**.
2. Select all the columns except Labels and click **Y, Columns**.
3. Click **OK**.
4. Press Ctrl, click the eth red triangle, and then select **Histogram Options > Show Counts**.

    Bar counts are added to all six histograms.
5. In the Distribution window, click the Distributions red triangle and select **Save Script > To Clipboard**.
6. Click a line or two after your `Open()` expression and select **Edit > Paste**.
7. Type a semicolon after the last close parenthesis.
8. Select **Graph > Scatterplot 3D**.
9. Select all the columns except Labels and click **Y, Columns**.
10. Click **OK**.
11. Copy and paste the script for Scatterplot 3D into the script window just like you did for your Distribution report. Be sure to add the semicolon at the end.
12. Select **Analyze > Multivariate Methods > Multivariate**.
13. Select all the columns except Labels and click **Y, Columns**.
14. Click **OK**.
15. Copy and paste the script for Multivariate into the script window just like you did for Distributions and Scatterplot 3D.

    You should see the following script:

```
dt = Open( "$SAMPLE_IMPORT_DATA/Solubil.xlsx" );
Distribution(
    Continuous Distribution( Column( :eth ), Show Counts( 1 ) ),
```

```
        Continuous Distribution( Column( :oct ), Show Counts( 1 ) ),
        Continuous Distribution( Column( :cc14 ), Show Counts( 1 ) ),
        Continuous Distribution( Column( :c6c6 ), Show Counts( 1 ) ),
        Continuous Distribution( Column( :hex ), Show Counts( 1 ) ),
        Continuous Distribution( Column( :chc13 ), Show Counts( 1 ) ),
    );
    Scatterplot 3D(
        Y( :eth, :oct, :cc14, :c6c6, :hex, :chc13 ),
        Frame3D( Set Grab Handles( 0 ), Set Rotation( -54, 0, 38) )
    );
    Multivariate(
        Y( :eth, :oct, :cc14, :c6c6, :hex, :chc13 ),
        Estimation Method( "Row-wise" ),
        Scatterplot Matrix( Density Ellipses( 0 ), Shaded Ellipses( 0 ) )
    );
```

Save the Script

You now have a script that reproduces all of the steps that you performed manually. Save the script, and close your data table and all its report windows.

1. In the script window that contains your script, select **File > Save** or **File > Save As**.

2. Specify a filename (for example, Weekly Report).

3. Click **Save**.

Run the Script

As long as your weekly updated Microsoft Excel file is saved in the same place and contains the same columns, you can run your script and automatically produce all your reports.

1. Open the script that you saved.

2. Select **Edit > Run Script**.

   Your Microsoft Excel file is opened in JMP, and all three of your reports appear.

You can send this script to others. As long as they have access to the same Microsoft Excel file in the same location, they can also run the script in JMP and see your reports.

Advanced Note: Auto-Submit

If you want a particular script to always be executed instead of opened into the script window, put the following command on the first line of the script:

```
//!
```

If this is not the very first line, with nothing else on the same line, this command does nothing.

You can override this command when opening the file.

1. Select **File > Open**.

2. Press Ctrl, select the JSL file, and then click **Open**.

    The script opens into a script window instead of being executed.

The command is also ignored when you right-click the file in the Home Window and select **Edit Script**.

Scripting Tools

Using the Script Editor, Log Window, Debugger and Profiler

JMP provides several programming tools for script writers. The script editor supports syntax coloring, autocompletes functions as you type, highlights matching braces, allows for code folding, and has additional features to help you develop scripts more quickly. Error messages and output are shown in the log window, which can be displayed inside the script editor. The JMP Scripting Language (JSL) Debugger and Profiler can help you troubleshoot your scripts.

Figure 4.1 Script Editor with Embedded Log and the Debugger
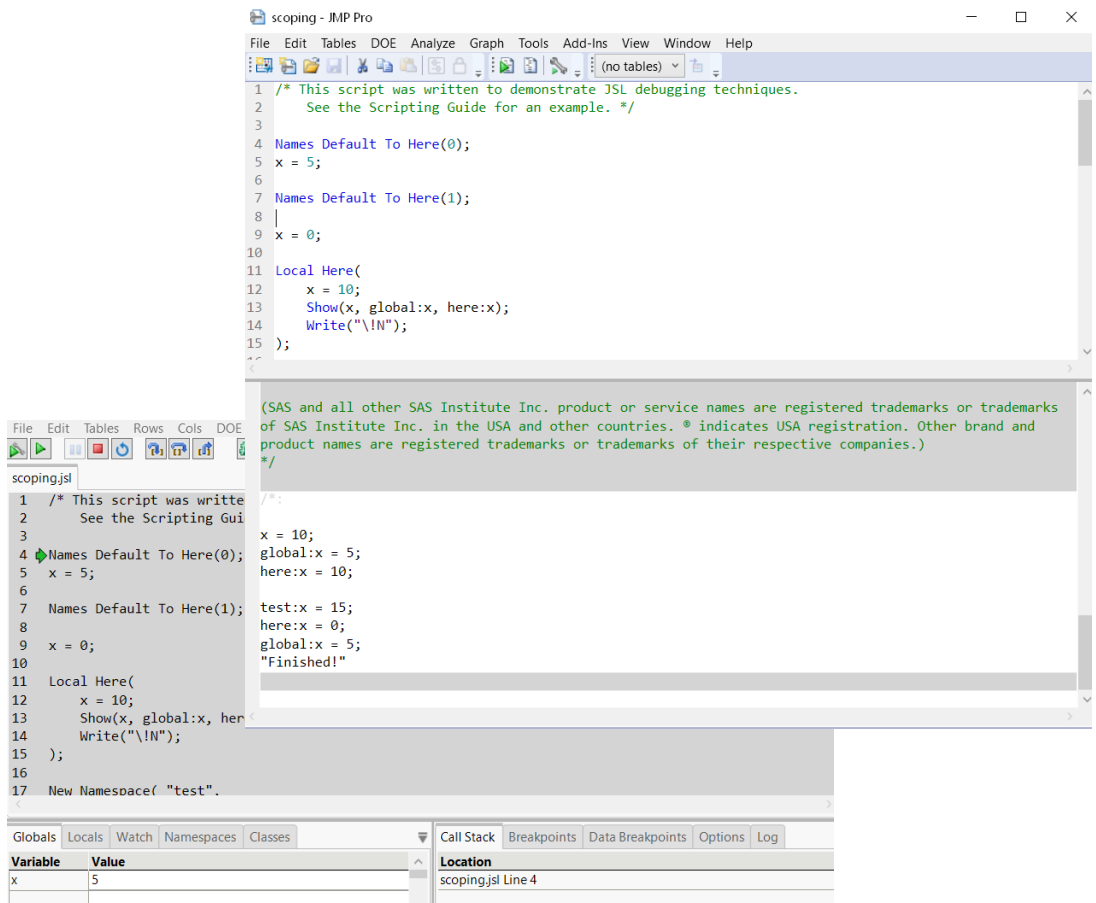
Contents

Work with the Script Editor

Introduction to the Script Editor

The script editor provides a friendly environment for writing and reading JSL scripts. Figure 4.2 shows basic features such as syntax coloring, inline commenting, and automatic formatting. Other common programming options are described later in this section.

Script editor features are also available in the log window and anywhere else that you can edit or write a script (for example, in the Scripting Index or Application Builder).

Figure 4.2 The Script Editor
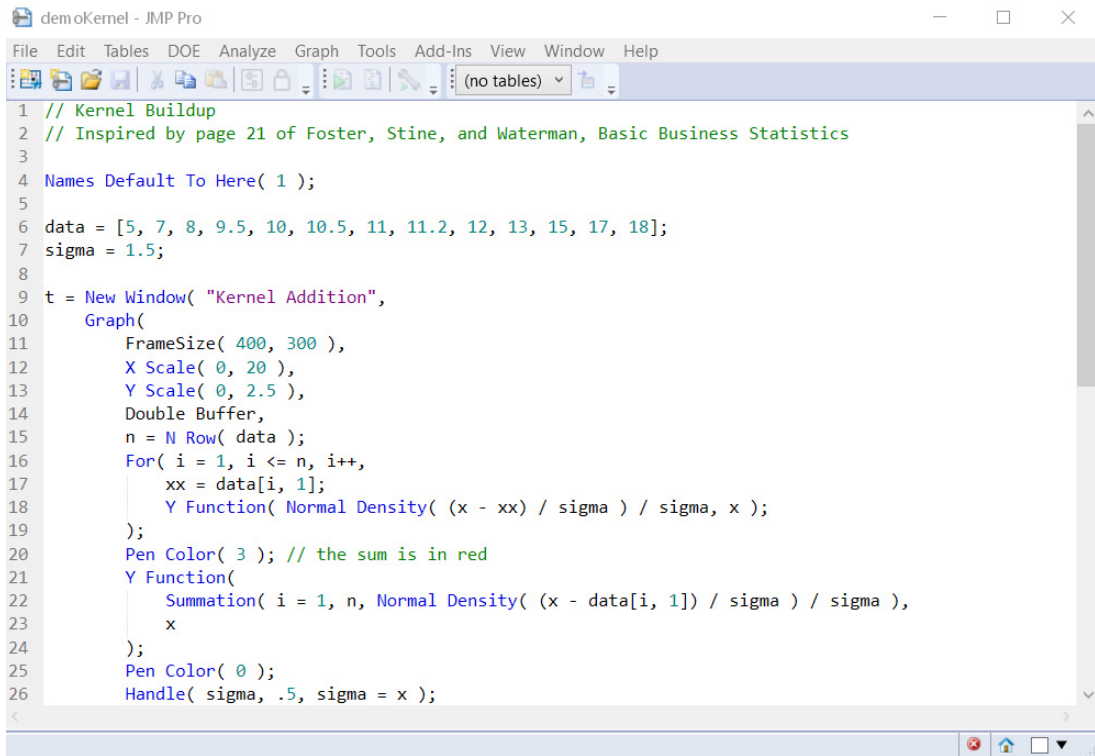
Tip: Consider setting the *Autosave timeout* value in the General preferences to automatically save open scripts at the specified number of minutes. This autosave value also applies to data tables, journals, scripts, projects, and reports.

Run a Script

To run an entire script, select **Edit > Run Script.**

To run specific lines in a script, select those lines and then select **Edit > Run Script**.

To run specific lines that are not adjacent, press Ctrl, select the lines, and then select **Edit > Run Script**.

On Windows, you can also click in a line or select several lines and press Enter on your numeric keypad.

Run the script automatically when you open it by using one of the following methods:

- Type `//!` on the first line.
- Include `Run JSL( 1 )` in the `Open()` statement:

```
Open( "$SAMPLE_SCRIPTS/scoping.jsl", Run JSL( 1 ) );
```

Stop a Script

To stop the script, press Esc on Windows (or Command-period on macOS). You can also select **Edit > Stop Script**. On macOS, **Edit > Stop Script** is available only when the script is running.

Edit a Script

To edit a script on Windows, press Insert. You can then type directly over (overwrite) any existing JSL code. Note that this feature is not available on macOS.

Color Coding and Themes

On Windows, the default font in the script editor is 10-point Consolas. On macOS, the default script editor font is 13-point Menlo.

JMP applies the following colors in the script window:

- black for text, identifiers (JSL functions), braces, and user macros
- white for the script editor background
- gray for the disabled background and guides
- green for comments
- purple for strings
- maroon for platform names

  Note: If a platform name is used in a message, the JSL messages color is used.

- teal blue for numbers
- dark blue for operator symbols, the first keyword, and JSL messages
- medium blue for function names, second and third keywords, and macros
- red for unknown objects

You can customize colors and fonts in the Customize Styles section of the Script Editor preferences. Syntax coloring for JSL, the log, SAS code, JavaScript, Text, JMP Text Analytics, R, Matlab, Python, C, JSON, and SQL are supported. In addition, options for dark and light themes are provided.

Autocomplete Functions

If you do not remember the exact name of a function, use autocompletion to see a list of functions that match what you have typed so far. Type part of the name, and then press Ctrl+Space on Windows (Option+Esc on macOS).

Suppose that you want to clear your JSL variables, but do not remember the command. You can type `clear` and then press Ctrl+Space, to see a list of possible clear commands. Select the command that you want to insert.

Figure 4.3 Autocomplete Example
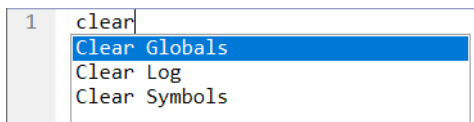
Tooltips

If you are using a function and do not remember the syntax or need more information about it, place the cursor over it to see a brief explanation. This works only with JSL function names, not platform commands, messages, or user-created functions. JSL function names are colored blue in the script editor.

The tooltip shows the syntax, arguments, and a brief explanation of the function (Figure 4.4). The tip also appears in the script editor window status bar.

Figure 4.4 Tooltip for a JSL Function
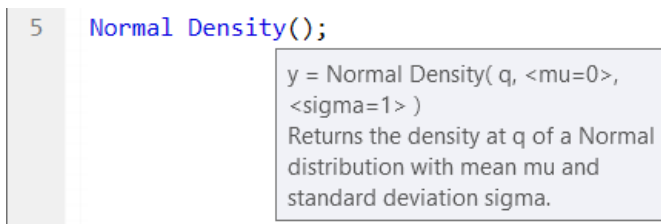
After running a script, you can also place the cursor over variable names to see their current value. To turn off variable tooltips, deselect **Preferences > Script Editor > Show Variable Value Tips**.

To turn off function tooltips, deselect **Preferences > Script Editor > Show Operator Tips**.

Tip: In addition to showing the tooltip for a function, you can display its Scripting Index entry. Run the script, press Alt, and then double-click the function. A description, sample script, and information about syntax is provided. You can also read help for the topic by running the script, right-clicking the function, and selecting **Help Browser**.

Example of a Tooltip for a JSL Variable

1. Enter and run the following line in a script window:

   ```
   my_variable = 8;
   ```

2. Hover over the variable name after you run the line.

   A tooltip shows the name of the variable and its value: 8.

3. Enter and run the following line:

   ```
   my_variable = "eight";
   ```

4. Hover over the variable name after you run the line.

   A tooltip shows the name of the variable and its value: "eight".

Split a Window

You can split the Script Editor window into two vertical or horizontal windows. This feature enables you to independently scroll through your code in two different places and edit the contents in both. When you make a change in one window, the change is immediately reflected in the other window.

• To split an open Script Editor window, right-click in the window and select **Split > Horizontal** or **Vertical**.

• To revert to a single window, right-click and select **Remove Split**.

Figure 4.5 Example of Splitting a Window Horizontally
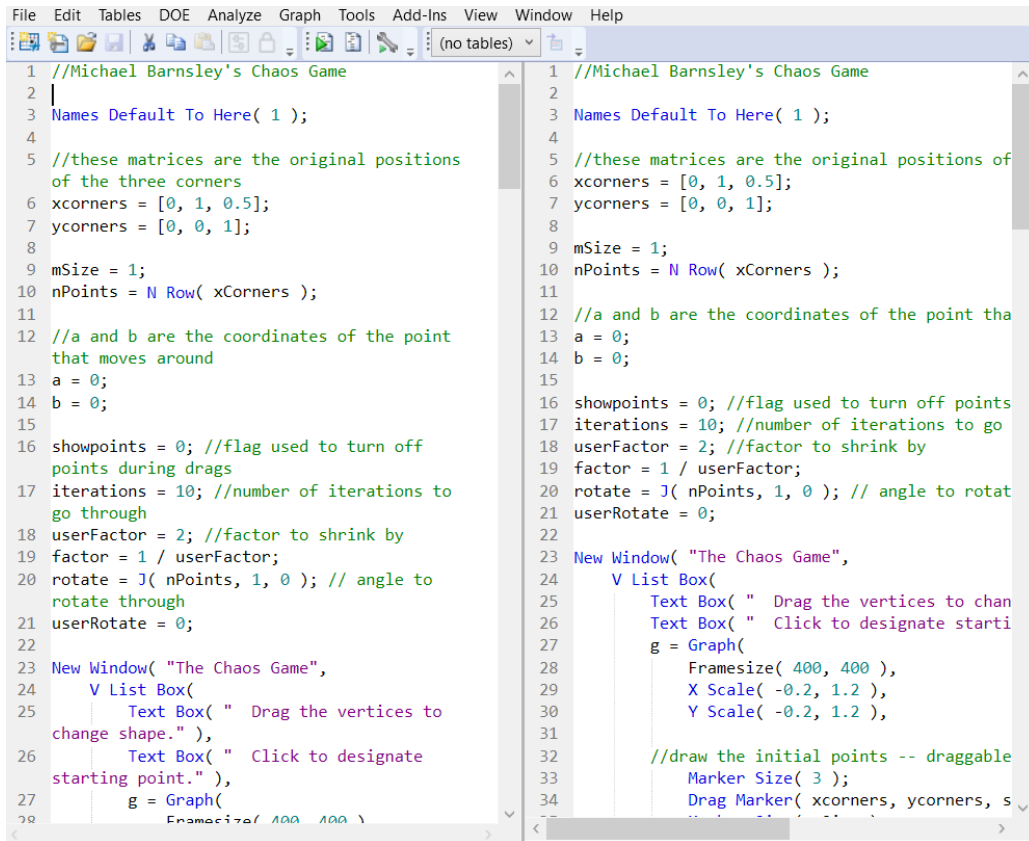
Match Parentheses, Brackets, and Braces

The script editor helps you match *fences* (or parentheses, square brackets, and braces) in the following ways:

- The matching closing fence is added when you type an opening fence.

- When you click next to either an opening or closing fence, the fence and its match are highlighted in blue. If the fence does not have a match, it is highlighted in red.

- If you double-click a fence, everything between the matching fence is selected (including the fences).

- If you put your cursor within an expression and press Ctrl+] on Windows (Command+B on macOS), the entire expression is selected. Fences that enclose the expression are included. Repeat this process to highlight the next-higher expression. Figure 4.6 shows an example.

Figure 4.6 Each Step in Matching Fences

```
<<Script(
    temp# = Num( tebAlpha# << Get Text );
    If( Not( Is Missing( temp# ) ) & (0 < temp# < 1),
        alpha# = temp#
    );
    tebAlpha# << Set Text( Char( alpha# ) );
    evalSummary#;
)
```

```
<<Script(
    temp# = Num( tebAlpha# << Get Text );
    If( Not( Is Missing( temp# ) ) & (0 < temp# < 1),
        alpha# = temp#
    );
    tebAlpha# << Set Text( Char( alpha# ) );
    evalSummary#;
)
```

```
<<Script(
    temp# = Num( tebAlpha# << Get Text );
    If( Not( Is Missing( temp# ) ) & (0 < temp# < 1),
        alpha# = temp#
    );
    tebAlpha# << Set Text( Char( alpha# ) );
    evalSummary#;
)
```
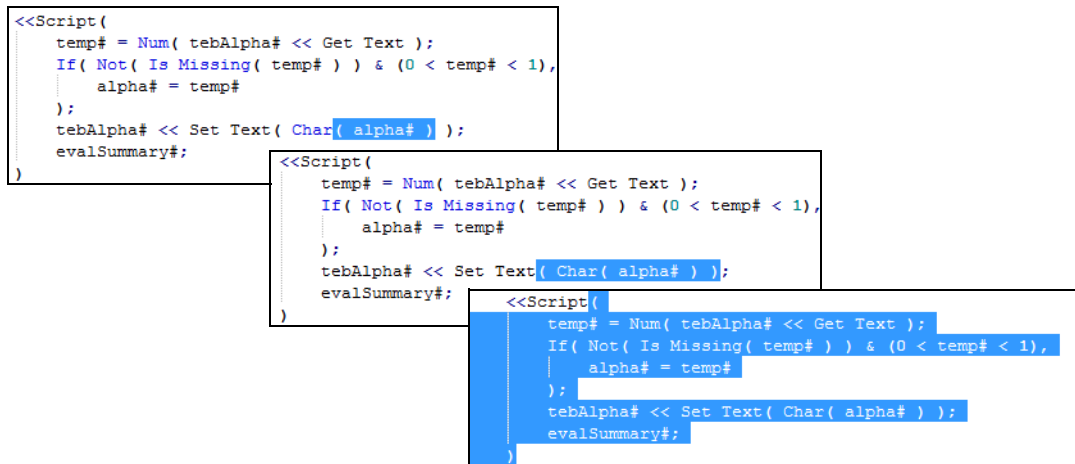
When you type an opening brace, JMP adds the closing brace. Enter code between the braces, type the closing brace, and then your cursor automatically moves after the closing brace that JMP added. This prevents you from accidentally adding an unnecessary closing brace.

You can turn on and off the autocompletion of braces in the JMP preferences. See "Set Preferences for the Script Editor" on page 70.

Select a Rectangular Block of Text

To select a rectangular block of text, press Alt and drag your cursor from the starting point to the end of the block. You can either copy or cut the text enclosed in the block.

Suppose that you want to the select all of the following code except for the comment marks.

```
// Y( :Y ),
// X( :X ),
```

Select a rectangular portion beginning with Y. When you paste, you get the following code:

```
Y( :Y ),
X( :X ),
```

The rectangular selection inserts returns where needed to maintain the structure of the text. Select `Get Menu Item State` on both lines in the following example.

```
bb << Get Menu Item State(1),
bb << Get Menu Item State(2),
```

When you paste, a return is inserted at the end of each line.

```
Get Menu Item State
Get Menu Item State
```

Select Fragmented Text

To select text that is not contiguous, press Ctrl on Windows or Command on macOS and drag your cursor over the text. Continue this action for any other text that you want to select. You can then copy and paste your selection into a new script or you can run the selected text. Text will be pasted or run in the order it was selected.

Drag and Drop Text

You can drag and drop text within a script editor window or between windows or from a data table into a script editor window. On Windows, pressing Ctrl before dragging and dropping copies the text. On macOS, the text is copied by default.

Drag and drop text in any of these ways:

- Select a row or column, pause, and then drop it into the script editor window.
- Double-click text in a text field and then drop it into the script editor window. Examples are text in a data table cell and any other selectable text.

On Windows, you can also drag and drop text into a minimized window.

1. Drag the text over the Home Window button ⬆ in the lower right corner of the window.

   The Home Window appears.

2. In the Home Window list, drag the text over the destination window.

   That window appears.

3. Drop the text where you want it.

Find and Replace

Many find and replace options are available in the script editor, including the support of regular expressions. For example, searching with the following regular expression:

    get.*name

returns messages such as "Get Button Name", and "GetFontName"

Basic regular expressions such as ^ and $ (which match the start of line and end of line) and \n (which matches a carriage return) are also supported.

See *Using JMP* for more information about the Search options.

Automatic Formatting

The script editor can format a script for easier reading. Any generated script (for example, by saving a platform script) is automatically formatted with tabs and returns in appropriate places.

You can also reformat individual scripts that are difficult to read (for example, scripts in which all commands are strung together with no whitespace characters). From the **Edit** menu, select **Reformat Script**.

Tip: This command alerts you if your script is badly formed (for example, if your parentheses are not matched).

Errors Are Marked by Special Characters

After you run a script, the /*###*/ characters in the log indicate the location of an error in the script execution. The following example shows a script in which the height column name is misspelled.

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:heihgt[3]; // misspelled column name
```

After you run the script, the log reads:

```
Scoped data table access requires a data table column or variable in
  access or evaluation of 'dt:heihgt' , dt:heihgt/*###*/
In the following script, error marked by /*###*/
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:heihgt/*###*/[3];
```

/*###*/ directly follows the misspelled height argument.

Typing in Multiple Rows at Once

When you edit a script, you can type in multiple rows at once instead of manually updating the same code on multiple lines. For example, in the following script, you want to change the s variable to aString. You can easily change the first instance of s to aString. To change the other instances at once, hold down Alt and select the column of s variables. Type aString once.

```
s = "";
For( i = 1, i <= n, i++,
  s += "Hello";
```

```
    s += "Number";
    s += Char( i );
);
```

The script then looks like this:

```
aString = "";
For( i = 1, i <= n, i++,
    aString += "Hello";
    aString += "Number";
    aString += Char( i );
);
```

Note: You can type in multiple rows at once, but you can type only in the same column. In the preceding example, the s variables are in the same column.

Add Code Folding Markers

You can add code folding markers that show the beginning and the end of the code block, enabling you to collapse and expand code inside stand-alone functions.

To turn on this feature, select **Code folding** in the Script Editor preferences. Then you can expand and collapse blocks of code by right-clicking on a script and selecting **Advanced > Expand All** or **Collapse All**.

Tip: To fold or expand all code in the entire script, press Shift+Ctrl and then click a code marker.

After you select this preference, Function and Expr expressions are foldable. See "Add More Folding Keywords" on page 69 for more information about adding folding markers to other expressions.

Figure 4.7 Code Folding Markers Shown in a Script

```
⊞computeBayes# = Expr(

  computeBayes#;

⊟updateBayes# = Expr(
    computeBayes#;
    ncb# << Set Values( factorProbability# );
    pcb# << Delete;
    tb# << Append( pcb# = Plot Col Box( "Probability", factorProbability# ) );
  );
```

By default, code does not remain collapsed after you save the script and restart JMP. To save the state of the folded code, select **Save and restore document state information** in the Script Editor preferences.

Add More Folding Keywords

Custom code folding is supported for other stand-alone functions as shown in the following example:

```
{"If", "For", "For Each Row", "While", "Try", "New Window", "V List Box",
  "H List Box"}
```

JMP supports multiple keyword lists. A system administrator can define a set of keywords in jmpKeywords.jsl and save the script in C:/ProgramData/SAS/JMP/ or designated directory listed below. You save your version of jmpKeywords.jsl in your C:/Users/<user>/Documents/ folder. JMP merges all keyword lists from the designated directories.

On Windows, the following directories are examined in the order listed:

- C:/ProgramData/SAS/JMP/16/
- C:/ProgramData/SAS/JMP/
- C:/Users/<user>/AppData/Roaming/SAS/JMP/16/
- C:/Users/<user>/AppData/Roaming/SAS/JMP/
- C:/Users/<user>/Documents/

On macOS, the following directories are examined in the order listed:

- /Library/Application Support/JMP/16/
- /Library/Application Support/JMP/
- ~/Library/Application Support/JMP/16/
- ~/Library/Application Support/JMP/
- ~/Documents/

Note that jmpKeywords.jsl is stored in the designated JMP directory, even if you are using JMP Pro.

Notes:

- The list in jmpKeywords.jsl is case insensitive.
- Code folding is not supported for messages, platforms, user-defined functions, and comments.
- After you edit and save the list in jmpKeywords.jsl, turn the **Allow additional code folding keywords** preference off and then back on for the changes to take effect. Messages in the log indicate that the keywords were loaded.

Advanced Options

Right-clicking on selected text in the Script Editor provides the following Advanced options:

Expand All   (Appears only if JSL code folding is on.) Expands all blocks of code.

Collapse All   (Appears only if JSL code folding is on.) Collapses all blocks of code.

Comment Block   Makes the selected text comments.

Uncomment Block   Uncomments the selected comments.

Make Uppercase   Changes all selected text to uppercase.

Make Lowercase   Changes all selected text to lowercase.

Set Preferences for the Script Editor

In the JMP preferences, customize the script editor settings such as the font, colors, and spacing options.

Setting the Fonts

1. Select **File > Preferences**.
2. Select the **Fonts** group.
3. Click **Mono** to set the font for the script editor.

For more information about font preferences, see *Using JMP*.

Set Script Editor Preferences

Select **File > Preferences > Script Editor** to further customize the editor. For more information about script editor preferences, see *Using JMP*.

Work with the Log

The Log in JMP has two modes:

- "Enhanced Log in JMP" on page 71
- "Text Log in JMP" on page 76

You can switch between the two modes by selecting **File > Preferences > Log**.
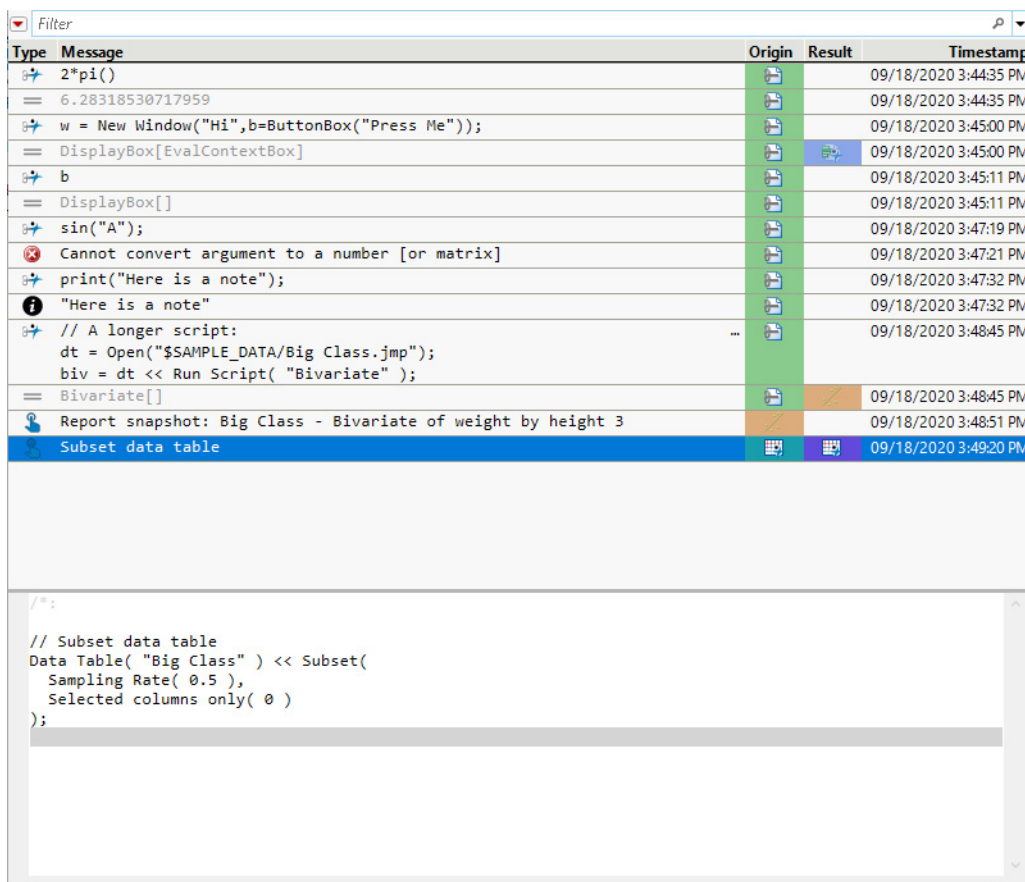
Note: The log shows the steps you take during data cleanup. For example, if you add a column property or import the data into Excel, then the JSL expressions for those actions are captured in the log. This enables you to clean up the data more efficiently when you receive a new data set.

Enhanced Log in JMP

The Enhanced Log is a tabular view of the events in a JMP session. To view the JMP Log, select **View > Log** (**Window > Log** on macOS). By default, the log in JMP appears in Enhanced mode. The log window contains events, or actions, in the rows of the table and the corresponding JSL code and output in a separate text pane. This tabular view of the events is easy to navigate, displays errors and warnings, and can help you create a JSL script of the actions in a JMP session. For an example of using the log to generate a JSL script for recorded actions, see "Example of Recording Actions in the Enhanced Log" on page 75.

Note: When you used the embedded log in a script window, the embedded log does not use the Enhanced Log mode. Further, any script output from a script window with an embedded log does not appear in the Enhanced Log.

Figure 4.8 Enhanced Log

Each row of the tabular view contains information in the following columns:

Type   Contains an icon that represents message type. For more information about the types of messages, see "Filter" on page 73.

> **Tip:** You can hover over an icon in the Type column to see the type of message that a particular icon represents.

Message   Contains a description of the event or result. Among other things, this could be a recorded action, JSL that was submitted, the result of JSL that was submitted, or an error message. Each description of a logged item or result is limited to three lines of text; longer items are denoted with an Ellipsis icon ⋯ in the upper right corner of the Message section of the row. The full text for longer items appears in the text pane at the bottom of the log window.

Origin   Contains an icon that shows the type of JMP window from which the logged item or result was generated. If the Color by Window option is selected in the red triangle menu next to Filter, logged items generated by the same JMP window are specified by a common color. You can click on the icon in the Origin column to bring the corresponding window to the front.

> **Note:** Not all rows in the log table have an icon in the Origin column.

Result   Contains an icon that shows the type of JMP window that contains the result. If the Color by Window option is selected, logged items generated by the same JMP window are specified by a common color. You can click on the icon in the Result column to bring the corresponding window to the front.

> **Note:** Not all rows in the log table have an icon in the Result column.

Timestamp   Contains a date-time value that indicates when the logged item was generated.

Enhanced Log Options

The red triangle menu in the Enhanced Log contains the following options:

Go To   (Available only when one or more rows of the log table are selected.) Brings the window corresponding to the selected row in the log table to the front. If an Origin window and a Result window both appear in the selected row, both windows are brought to the front with the Result window on top of the Origin window.

Run Scripts   (Available only when one or more rows of the log table that contain a scriptable action are selected.) Runs the JSL script that corresponds to the selected rows in the log table.

Color by Window   Specifies whether the rows in the log table are colored by the JMP window that created the row of the log output.

Clear   (Available only when one or more rows of the log table are selected.) Removes the selected row or rows from the log table.

Caution: This action cannot be undone.

Clear Log   Removes all rows from the log table.

Caution: This action cannot be undone.

Filter   Contains the following options to filter what types of logged events appear in the log table.

Result =   Shows or hides rows that correspond to the results of running JSL.

Script ⇢   Shows or hides rows that correspond to submitted JSL.

Log ❶   Shows or hides rows that correspond to informational log messages.

Warning ⚠   Shows or hides rows that correspond to warning messages in the log.

Error ⊗   Shows or hides rows that correspond to error messages in the log.

Action ♨   Shows or hides rows in the log table that correspond to interactive JMP actions. These actions include data table operations, platform launches, and closing report windows. You can customize which actions are shown in the log table by selecting specific options in the Action Recording submenu.

Action Recording   Contains the following options to filter what types of interactive JMP actions appear in the log table.

Data Table Operations   Shows or hides rows that correspond to manipulations of a data table. These actions can include row and column operations, as well as actions in the Tables, Rows, and Cols menus.

Platform Launch   Shows or hides rows that correspond to launching a JMP platform.

Report Snapshot on Close   Shows or hides rows that correspond to a JSL script that is saved when a report window is closed.

Save Script   Contains the following options that save a script that reproduces all selected rows in the log table. If no rows are selected, these options save a script to reproduce all events recorded in the log table.

To Script Window   Saves the script to a script window.

To Clipboard   Saves the script to the clipboard.

Make Into Data Table   Creates a data table that contains all rows from the log table. If rows are selected in the log table, they are also selected in the data table that is created.

Filter the Results in the Enhanced Log

To the right of the red triangle menu, use the search field to filter the items in the log table view.

Click the down arrow button next to the search box to refine your search.

Contains Terms   Returns items that contain a part of the search criteria. A search for "ease oom" returns messages such as "Release Zoom".

Contains Phrase   Returns items that contain the exact search criteria. A search for "text box" returns entries that contain "text" followed directly by "box" (for example, "Context Box" and "Text Box").

Starts With Phrase   Returns items that start with the search criteria.

Ends With Phrase   Returns items that end with the search criteria.

Whole Phrase   Returns items that consist of the entire string. A search for "text box" returns entries that contain only "text box".

Regular Expression   Enables you to use the wildcard (*) and period (.) in the search box. Searching for "get.*name" looks for items that contain "get" followed by one or more words. It returns "Get Color Theme Names", "Get Name Info", and "Get Effect Names", and so on.

Invert Result   Returns items that do not match the search criteria.

Match All Terms   Returns items that contain both strings. A search for "t test" returns elements that contain either or both of the search strings: "Pat Test", "Shortest Edit Script" and "Paired t test".

Ignore Case   Ignores the case in the search criteria.

Match Whole Words   Returns items that contain each word in the string based on the Match All Terms setting. If you search for "data filter", and Match All Terms is selected, entries that contain both "data" and "filter" are returned.

Example of Recording Actions in the Enhanced Log

You can use the Enhanced Log to record many of the interactive actions that you perform for data cleanup and analysis. In this example, the Enhanced Log is used to generate a JSL script for some data manipulation and analysis actions.

1. Select **View > Log**.
2. Click the red triangle next to the Filter box and select **Clear Log**.
3. Select **Help > Sample Data Library** and open Car Physical Data.jmp.
4. Right-click Displacement in the Columns panel in the data table, and select **Column Info**.
5. Type Engine Size for the Column Name. Click **OK**.
6. Right-click Type in the Columns panel in the data table, and select **Column Info**.
7. Select Ordinal for the Modeling Type. Click **OK**.
8. Click Horsepower in the Columns panel in the data table and drag it above Weight.
9. Select **Graph > Graph Builder**.
10. Drag Horsepower to the **Y** drop zone.
11. Drag Engine Size to the **X** drop zone.
12. Close the Graph Builder window.
13. Select **Analyze > Distribution**.
14. Select Type and Horsepower and click **Y, Columns**.
15. Click **OK**.
16. Click the red triangle next to Horsepower and select **Continuous Fit > Fit Lognormal**.
17. Close the Distribution window.
18. Select **View > Log**.

Figure 4.9 Enhanced Log

19. Press Ctrl and select the first, second, third, and last rows of the log table.

20. Click the red triangle next to the Filter box and select **Save Script > To Script Window**.

The script to reproduce the selected actions appears in a new script window. You can save this script to reuse later or use it as a starting point for a more complex analysis.

Text Log in JMP

The Text Log in JMP is a text-based log of JSL scripts that have been run during a JMP session and the results of those scripts. When the log is open, output appears in the log after you run a script. The script that was run is shaded in the log, and the corresponding output appears beneath it. To view the log in Text mode, select Text for the Mode preference after selecting **File > Preferences > Log**.

Syntax and compatibility errors are reported in the log, including the line number, script filename, and code that JMP could not process. Many interactive JMP actions are also recorded in the log.

Tips:

- To omit compatibility warnings from the log, deselect **Show log warnings for JSL compatibility changes in 12** in the JMP General preferences.

- To choose which actions are recorded in the log, you can use the Action Recording preferences after selecting **File > Preferences > Log**.

Show the Text Log in the Script Window

You can view the log inside the script window by right-clicking and selecting **Show Embedded Log**. To view the embedded log by default, select **Show embedded log on script window open** in the Script Editor preferences. This option makes it easy to edit and run a script, quickly see the results of your changes, and then continue to develop the script. The embedded log uses the Text log mode. See "Text Log in JMP" on page 76.

The embedded log always appears in the Scripting Index script window, but it is not available in Application Builder and the Debugger.

Figure 4.10 The Script Window with an Embedded Text Log
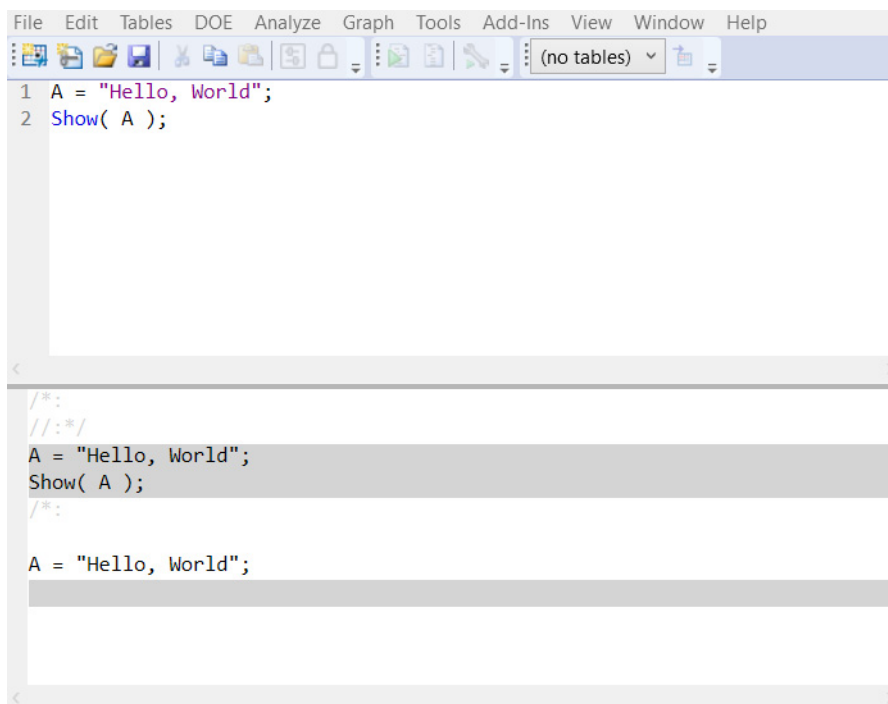
Debug or Profile Scripts

In an open script, click the **Debug Script** button ![debug icon] (or select **Edit > Debug Script**) to show the script in the JSL Debugger window. You can also use a keyboard shortcut:

- Press Ctrl+Shift+R (Windows).
- Press Shift+Command+R (macOS).

The JSL Debugger helps identify the point at which a script causes an error or fails. Rather than commenting out portions of the script or adding `Print()` expressions, you can use the Debugger to find the problem.

Once the JSL Debugger appears, you can continue in this mode, or you can click the **Profile JSL Script** button 🐞 to move into the JSL Profiler mode. The JSL Profiler helps with optimization. You can profile your scripts during execution to see how much time is spent executing particular lines or how many times a particular line is executed.

Tip: To debug a script automatically when you open it, include `Debug JSL( 1 )` in the `Open()` statement:

```
Open( "$SAMPLE_SCRIPTS/scoping.jsl", Debug JSL( 1 ) );
```

Debugger and Profiler Window

The Debugger opens in a new instance of JMP (Figure 4.11). The original instance is inoperable until the script produces something that requires interaction. At that point, the Debugger window becomes inoperable until you perform whatever action is required. Then control is returned to the Debugger. Close the Debugger to work again in the original instance of JMP.

Figure 4.11 The Debugger Window
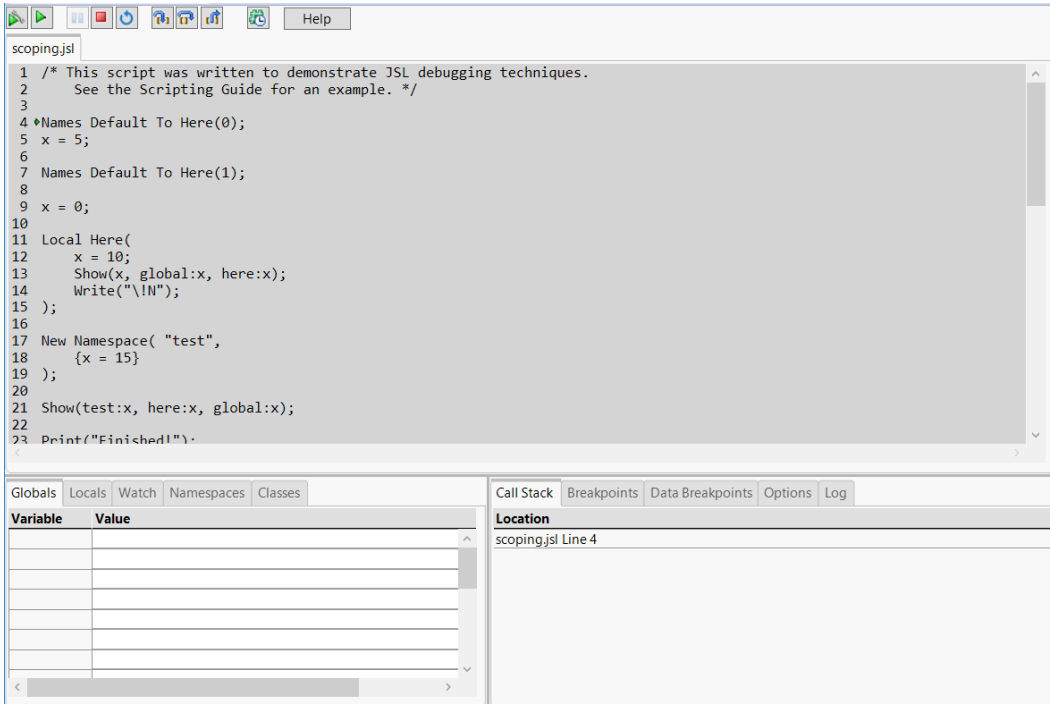
Use the buttons at the top to control the Debugger or the JSL Profiler. One or more scripts that you are debugging or profiling are shown in tabs. If your script includes other scripts, each one opens in a new tab.

Tabs in the bottom portion of the Debugger provide options to view variables, namespaces, the log, and the current execution point; work with breakpoints; and set options.

Tip: To view or edit long values on a tab in the Debugger, right-click the value and select **Edit**. The code opens in a script editor.

Using the Execution Buttons

Use the buttons at the top to control execution of the script within the Debugger or JSL Profiler.

Table 4.1 Description of the Debugger Buttons

Button	Button Name	Action
	Run	Runs the script in the Debugger until it reaches either a breakpoint or the end of the script.
		Note: If there is an error in the script, an abbreviated error message is displayed at the top of the window. Click **More** to see the complete error message.
	Run without breakpoints Run profiler	Runs the script through the end without stopping.
	Break All	If the script is busy, click Break All to stop all action in the script and return to the Debugger or JSL Profiler (for example, if you are in a very long loop).
		The Debugger or JSL Profiler might not be able to break execution if the executing script is waiting on some interactive user action, such as completing a window or interacting with an opened window.
	Stop	Stops debugging the script and exits the Debugger or the JSL Profiler.
	Restart	Closes the current Debugger session and opens a new session.

Table 4.1 Description of the Debugger Buttons *(Continued)*

Button	Button Name	Action
	Step Into	Lets you step into a function or an included file. Otherwise, it behaves the same as Step Over.
	Step Over	Runs all expressions on a single line, or a complex expression that spans multiple lines, without stepping into a called expression, function, or Include() file.
	Step Out	Runs the current script or function to a breakpoint or the end and returns to the calling point. If you are in the main script and the Debugger reaches the end, a message appears: Program execution terminated. The Debugger remains open in order for you to inspect the final program conditions.
	Profile JSL Script	Opens the JSL Profiler. (Press the Run profiler button to start the JSL Profiler.) Use the JSL Profiler to see how much time is spent executing particular lines or how many times a particular line is executed. **Notes:** - You can switch back and forth between the Debugger and JSL Profiler modes only prior to the start of the program. - Some of the debugger buttons are disabled when profiling. - All breakpoints are disabled when running in the JSL Profiler mode.
	Show Profile by Line Count	Shows the number of times each line is executed.
	Show Profile by Time	Shows how much time is spent executing a line.
	Show Profile by Count	For line counts, shows the number of times the line is executed. For time, shows the number of microseconds (or milliseconds or seconds) the line takes to complete.
	Show Profile by Percent	For line counts, shows the individual line count divided by the total line count. For time, shows the percentage of time spent on an individual line (line time/total time*100).
	Time Units	Sets the time unit to microseconds, milliseconds, or seconds. Available in the JSL Profiler after you click the Run profiler button ▶ .

Table 4.1 Description of the Debugger Buttons *(Continued)*

Button	Button Name	Action
Red ▾	Color Theme	Sets the color of the shading for the JSL Profiler. Available in the JSL Profiler after you click the Run profiler button ▶ .

Variable Lists

The tabs on the bottom left of the Debugger let you examine global variables, local variables, watch variables, and variables within namespaces.

Globals   The Globals tab lists all global variables and updates their values as you step through the script. Each variable is added as it is initialized. If there are already global variables defined from running earlier scripts, they will be listed with their current values when you start the Debugger. See "View Variables" on page 85.

Locals   The Locals tab lists all variables by scope and updates their values as you step through the script. Select a scope in the menu. See "View Variables" on page 85.

Watch   If there is a particular variable or value of an expression whose values you want to watch as you step through the code, you can add them here. This is particularly useful if your script uses many variables that might be difficult to watch in the Globals or Locals lists. See "Work with Watches" on page 85.

Namespaces   As namespaces are defined, they are added to the menu. Select a namespace to view any variables and their values used within the namespace. See "View Variables" on page 85.

Classes   The Classes tab lists classes that are defined in the script. See "Classes" on page 297 in the "Programming Methods" chapter.

Debugger Options

The tabs on the bottom right of the Debugger let you view the call stack, work with breakpoints, set options, and view the log.

Call Stack   The call stack lists the current execution point in scripts and functions. The main script is always the first script listed. If you call a function, the function is added on top of the calling script. Likewise, any included files are added to the top of the list as you step through them. When you exit any function or script, it is removed from the list and you return to the next one in the list. The current line numbers are updated as you step through.

Double-click a row in the call stack to move the cursor to the specified line.

Breakpoints   Add, edit, delete, and disable or enable breakpoints on lines. See "Work with Breakpoints" on page 82. You can also double-click a row on the Breakpoints tab to move the cursor to the specified line.

Data Breakpoints   Add, edit, delete, and disable or enable breakpoints in variables. Data breakpoints watch for changes in variables, unlike breakpoints, which watch for changes in lines.

Options   Set the Debugger preferences interactively on this tab. See "Modify Preferences in Debugger" on page 86.

Log   The log from the script that you are debugging is shown on this tab.

Work with Breakpoints

A *breakpoint* interrupts the execution of a script. Although you can step through a script line by line, this can be tedious and lengthy for a long or complex script. You can set breakpoints at places of interest and simply run the script in the Debugger. The script is run normally until a breakpoint is reached. At the breakpoint, the Debugger stops executing the script so that you can look at the values of variables or start stepping line by line.

Breakpoints that appear on the Breakpoints tab in the lower right corner watch for changes in a line. Breakpoints that appear on the Data Breakpoints tab watch for changes in variables.

JMP preserves breakpoints across sessions. So when you close and reopen JMP, the breakpoints still appear.

Tip: Turn on line numbers by right-clicking in the script and selecting **Show Line Numbers**. You can also show line numbers by default in all scripts by modifying the Script Editor preferences.

Create a Breakpoint

When creating a breakpoint, you can specify settings such as conditions and break behavior. To do so, click ▣ on the Breakpoints tab or Data Breakpoints tab, and then enter the breakpoint information.

Otherwise, create a quick breakpoint by doing one of the following:

• In the Debugger margin, click the appropriate line (to the right of the line number if displayed).

• In the Debugger margin, right-click in the margin where you want the breakpoint and select **Set Breakpoint**.

The red breakpoint icon appears where you inserted the breakpoint and on the Breakpoints tab.

Delete Breakpoints

Do one of the following:

- In the Debugger margin, click the breakpoint icon.
- In the Debugger margin, right-click the breakpoint icon and select **Clear Breakpoint**.
- On the Breakpoints tab, select the breakpoint and then click ✖.
- On the Breakpoints tab, click ✖ to delete all breakpoints (not just the selected breakpoints).

The red breakpoint icon is removed, and the breakpoint no longer appears on the Breakpoints tab.

Disable and Enable Breakpoints

Disabling a breakpoint is helpful when you potentially fix a problem and then want to see whether the script will run correctly past that breakpoint. You can then enable the breakpoint when necessary rather than re-creating it.

Do one of the following:

- In the Debugger margin, right-click the breakpoint icon and select **Enable Breakpoint** or **Disable Breakpoint**.
- On the Breakpoints tab, select or deselect the breakpoint's check box.
- On the Breakpoints tab, click 🔴 to disable or enable all breakpoints.

A disabled breakpoint turns white; enabled breakpoints are shaded red.

Specify and Clear Conditional Expressions on Breakpoints

Setting a condition on a breakpoint is an alternative to single-stepping through code. Rather than single-step and view the variables for each expression, you specify that the script break only when a condition is met. Then you can step through the code and figure out where the problem arises.

Suppose that a calculation in your script is incorrect, and you suspect the problem occurs when $i==19$. Set a conditional breakpoint for $i==18$. The Debugger will run until that condition is met, then you can step through the code to identify the problem.

Specify a Breakpoint Condition

1. Right-click the breakpoint icon and select **Edit Breakpoint**.
2. On the Condition tab, select **Condition** and enter the conditional expression.
3. Specify whether to break when the expression **Is true** or **Has Changed**.
4. Click **OK**.

Disable or Enable a Condition

1. Right-click the breakpoint icon and select **Edit Breakpoint**.

2. On the Condition tab, deselect or select **Condition**.

Delete a Condition

On the Breakpoints tab, click in the breakpoint's Condition field and press Delete.

Specify Break Options

Right-clicking the breakpoint and selecting **Edit Breakpoint** provides a quick way to manage breakpoint behavior. Alternatively, select the breakpoint on the Breakpoints tab and click . Both methods display the Breakpoint Information window, where you customize settings on the Hit Count and Action tabs.

Change the Hit Count

You can control the number of times a breakpoint must be hit and when the break occurs. For example, to break when the condition is met twice, select **break when the hit count is equal to** and type 2 on the Hit Count tab.

Define an Action

You also have the option of defining a JSL expression or script that the Debugger executes when a breakpoint is hit and execution has stopped. This script is called an *action*. On the Action tab, enter the JSL expression to be executed.

Run the Script to the Cursor

When you right-click and select **Run To Cursor**, all expressions before the location of the cursor are executed. Select this option when you want to see values only up to the current line. To see values when each expression is executed, use the stepping options.

Tips for Setting Breakpoints

- If you do not want to watch for errors in a specific loop, set a breakpoint after the loop ends. The Debugger will hit the next breakpoint rather than stepping through each line of the loop.

- Avoid inserting a breakpoint in lines that do not trigger an action (such as comments, blank lines, and end parentheses). Debugger will not break on these lines.

- When you insert breakpoints, close Debugger, and edit the script, the breakpoints remain on the original line numbers. You might need to delete and then reinsert the breakpoints.

- Breakpoints are remembered across Debugger sessions. This means that your breakpoints list includes breakpoints that have been set in all scripts, not just the script that you are currently debugging.

- Breakpoints are remembered by the Debugger session, not by each script. This means that breakpoints are listed even for scripts that have been moved or deleted.

- On the Breakpoints tab, click ✖ to remove all breakpoints in scripts whether they are currently open or not, or for scripts that no longer exist.

View Variables

The variables lists are populated as they appear in the script. Their values are updated every time the script changes them. If you are uncertain why a variable has a particular value when you run your script, you can watch its value at every step to see what happens.

You can also assign the variables whatever value you want. For example, if you are stepping through a For() loop but are interested only in what happens starting with a particular iteration, you can assign your iterating variable that value. Step through the first part of the loop that initializes the iteration variable and then assign it the value that you want in the variable list at the bottom. Then when you step through, the loop begins executing at that point.

Tips for Managing Variables

- If you have run several scripts using the global scope, you might want to clear or delete global variables. This makes the list of variables in the Debugger shorter and relevant. Use the Delete Symbols() function to do so. You can also close JMP and restart to clear the space.

- If your script uses so many variables that they are difficult to find and watch in the variable lists, add watches for the specific variables in which you are interested.

- To view or edit long values on a tab in the Debugger, right-click the value and select **Edit**. The code opens in a script editor.

Work with Watches

JMP preserves the Watch variables across sessions. So when you close and reopen JMP, the Watch variables are still listed on the Watch tab.

Create a Watch

- On the Watch tab, click ➕ and enter the value in the window.

- In the Debugger, right-click the line that you want to watch, select **Add Watch**, and then enter the variable name in the window.

- In the Debugger, place the cursor in or next to a variable name (or select the variable name), right-click, and select **Add Watch**.

- On the Watch tab, enter the variable in an empty **Variable** field.

Modify a Watch

Do one of the following on the Watch tab to enter a new value:

- Select the watch and click ▣.
- Click in the Variable field and enter a new value.

Delete Watches

Do one of the following on the Watch tab to delete watches:

- Select the watch and then click ✖.
- Click ✖ to remove all watches.

Modify Preferences in Debugger

The Debugger lets you change preferences as you work in the Debugger. Select the Options tab to find the following settings:

Show Line Numbers   Shows or hides the line numbers for the script in the Debugger.

Break on Multiple Statements Per Line   Stops executing the script between each expression in a single line.

Break On Throw   Breaks when the script executes the Throw() function. For example, Throw() might be enclosed in a Try() expression. The Debugger breaks on Throw() instead of continuing through the rest of the expression. This lets you identify where the problem occurred in the script and then return to debugging.

Break On Execution Error   Stops executing the script when the error occurs rather than closing the Debugger.

Warn On Assignment In Condition   Shows a warning when you enter a breakpoint condition that contains the assignment. For example, if you have a breakpoint on x = 1 and add the condition x = 1 to the breakpoint, you are prompted to verify the assignment of x.

Enter Debugger Upon Termination   Keeps the Debugger open after a JSL program terminates execution. On by default, this option lets you examine attributes of the executed program.

Break For Compatibility Warnings   Breaks when a JSL compatibility issue is found.

Persistent Debugger Sessions

JMP saves all breakpoints and watches until you delete them. Other user-specific settings, such as column widths on the tabs and the Debugger window size, persist between sessions of JMP.

These settings are stored in a file named JMP.jdeb, the location of which is defined in the USER_APPDATA variable:

- Windows 7 and higher: "/C:/Users/<username>/AppData/Roaming/SAS/JMP/16/"
- macOS: "/Users/<username>/Library/Application Support/JMP/16/"

As usual, the values of local variables, global variables, and namespaces clear when you close and reopen JMP.

Note: On Windows, the paths differ based on the JMP edition. In JMP Pro, the path refers to "JMPPro".

Examples of Debugging and Profiling Scripts

This section includes examples of setting breakpoints to watch variables; stepping into, over, and out of expressions; watching variables in different scopes and namespaces; debugging interactive scripts; and profiling scripts with the JSL Profiler.

Example scripts are located in the Samples/Scripts folder.

Tip: Make sure that **Show Line Numbers** is selected on the Debugger Options tab before proceeding.

Using Breakpoints and Watching Global Variables

The following example shows how to set a breakpoint in a loop and watch variables change through each iteration of the loop.

1. Open the string.jsl sample script and click the **Debug Script** button.
2. Click in the margin for line 12 to add a breakpoint.

   You should have a breakpoint for the following expression inside the For() loop:

   ```
   stringFunction(i);
   ```

Figure 4.12 Set the Breakpoint

```
string.jsl
3
4 ▶ stringFunction = Function( {i}, {y},
5        y = Char(i);
6        str = str || y;
7    );
8
9    str = "";
10
11   For( i = 0, i <= 9, i++,
12 ●     stringFunction(i);
13   );
14
15   Print( "Finished.", str );
```

3. Click **Run**.

    The first two expressions are evaluated:

    –   stringFunction is defined as a function.

    –   str is defined as an empty string.

    Both variables and their types and values have been added to the Globals list. In addition,
    the For() loop has been evaluated up to the line with the breakpoint, shown in
    Figure 4.12.

    –   i has been assigned to 0.

    –   i and its value and type have been added to the Globals list.

    –   i has been determined to be less than or equal to 9.

    –   stringFunction() has not yet been called.

Figure 4.13 View the Initial Global Variables

Globals	Locals	Watch	Namespaces	Classes

Variable	Value	Type
i	10	Number
str	"0123456789"	String
stringFunct...	Function({i}, {y}, y = Char(i); str = str || y;)	Function

Tip: The initial value of i is 10 because the script was run before entering the Debugger; it's
showing the value of i after running the script (10). Start a new instance of JMP, open the
script without running it, and enter the Debugger immediately.

4. Click **Run** again.

    The script runs until it hits the breakpoint. The results are shown in Figure 4.14.

    –   stringFunction() is called, evaluated, and returns to the loop.

    –   i is incremented and determined to be less than or equal to 9.

– In the Globals list, i is now 1 and str is now "0".

Figure 4.14 Global Variables at First Breakpoint

Globals	Locals	Watch	Namespaces	Classes	
Variable	**Value**				**Type**
i	1				Number
str	"0"				String
stringFunct...	Function({i}, {y}, y = Char(i); str = str || y;)				Function

5. Click **Run** again.

The script runs until it hits a breakpoint. The results are shown in Figure 4.15.

– stringFunction() is called, evaluated, and returns to the loop.

– i is incremented and determined to be less than or equal to 9.

In the Globals list, i is now 2, and str is now "01".

Figure 4.15 Global Variables at Second Breakpoint

Globals	Locals	Watch	Namespaces	Classes	
Variable	**Value**				**Type**
i	2				Number
str	"01"				String
stringFunct...	Function({i}, {y}, y = Char(i); str = str || y;)				Function

You can continue to click **Run** and watch i and str change with each iteration of the loop. Or, click **Run without breakpoints** to complete running the script and exit the Debugger.

Stepping Into, Over, and Out

Step Into, **Step Over**, and **Step Out** offer flexibility when your script contains expressions, functions, or includes other JSL files.

1. Open the scriptDriver.jsl sample script and click the **Debug Script** button.

2. This script writes information to the log, so select the Log tab at the bottom of the Debugger to view the messages.

3. Click **Step Over**.

The first line in the script is evaluated.

4. Click **Step Over** again.

The current expression is evaluated, and the Debugger moves to the following line. In this case, the expression is a few lines long, and it assigns an expression to a variable.

5. Click **Step Over** again.

This expression is several lines long, and assigns a function to a variable.

Line 30 calls the expression that was created earlier.

6. Click **Step Over**.

The Debugger steps into the expression, running it line by line.

7. Continue clicking **Step Over** until the expression ends.

The Debugger returns to the line following the expression call.

Line 31 calls the function defined earlier.

8. Click **Step Over** to run the function without stepping into it. The Debugger runs the entire function, and returns to the line following the function call.

Line 33 includes another script.

9. Click **Step Into**.

The Debugger opens the script in another tab and waits.

10. Click **Step Over**.

The next line in the included script is run.

11. Click **Step Out**.

The Debugger runs the rest of the included script and returns to the line following the `Include()` function.

Watching Variables in Different Scopes and Namespaces

Tabs at the bottom of the Debugger window let you watch variables as they are created and changed. This example shows variables in several scopes and a namespace.

1. Open the scoping.jsl sample script and click the **Debug Script** button.

2. Click **Step Over**.

The fourth line turns off `Names Default To Here`. If you run this script again in the same JMP session, this line resets the scoping so that the first variable that is created is in the global scope.

3. Click **Step Over**.

A global variable named *x* is created. On the Globals tab, *x* has been added to the list, showing its value as 5 and its type as number.

4. Select the **Locals** tab, and then select **Global** from the list of scopes.

The global variable *x* is also shown here.

5. Click **Step Over** twice.

`Names Default To Here` is turned on, which places the rest of the script into a Here scope. Then a new variable *x* is created in that scope.

Notice that the value of the global variable *x* has not changed.

6. Select **Here** from the list on the Locals tab.

    The local *x* is listed under **Here**, with its value and type.

7. Click **Step Over**.

    A `Local Here` scope is created. A second Here scope is shown in the Locals list.

8. Click **Step Over**.

    A new *x* variable is created in this Here scope. On the Locals tab, select each of the three scopes from the list (**Here**, **Here**, and **Global**) to see three different *x* variables.

9. Click **Step Over**.

    Look in the Debugger's log to see the output. Notice that *here:x* scopes to the local here, not the script window's here.

10. Click **Step Over**.

    After writing an empty line to the log, the script exits the Local Here scope. The second Here, along with its' *x* variable, has disappeared from the Locals list.

11. Click **Step Over**.

    A namespace called "test" is created, with another variable named *x*. Select the **Namespaces** tab to see it.

12. Click **Step Over** and look at the log.

13. Click **Step Over** to exit the Debugger.

Using the Debugger with Interactive Scripts

When your script creates interactive elements, the Debugger hands control back to the main instance of JMP so that you can interact with it. When you are finished, control returns to the Debugger.

1. Open the interactive.jsl sample script and click the **Debug Script** button.

2. Click **Step Over** twice.

    The `New Window` expression is evaluated, and a modal window waiting for input is created. You might need to move the Debugger window to see the new modal window.

3. Enter two numbers in the Assign X and Y window and click **OK**.

    Control is given back to the Debugger.

4. Click **Step Over** three times and look at the log in the Debugger.

    The log shows the two new numbers that you entered in the window.

5. Click **Step Into** to exit the Debugger.

Using the JSL Profiler

Use the JSL Profiler to see how much time is spent executing particular lines or how many times a particular line is executed.

1. Open the string.jsl sample script.
2. Click the **Debug Script** button ![debug icon].
3. Click the **Profile JSL Script** button ![profile icon].

Figure 4.16 Initial JSL Profiler Window


```
string.jsl

 1 /* This script was written to demonstrate using breakpoints and watching variables
 2    in the JSL Debugger. See the Scripting Guide for an example. */
 3
 4 stringFunction = Function( {i}, {y},
 5     y = Char(i);
 6     str = str || y;
 7 );
 8
 9 str = "";
10
11 For( i = 0, i <= 9, i++,
12     stringFunction(i);
13 );
14
15 Print( "Finished.", str );
16
```
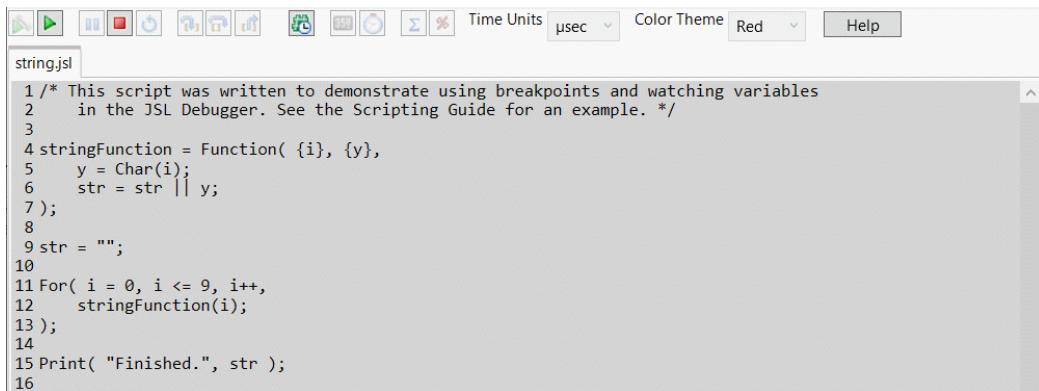
4. Click the **Run** button ![run icon] to start profiling.

   The profiler collects information about the number of times a statement is executed and the time it takes to execute it. Time is cumulative and collected each time a JSL statement is executed.

Figure 4.17 Profiled Script Window


```
string.jsl

 3
 4 83.989% stringFunction = Function( {i}, {y},
 5  0.911%     y = Char(i);
 6  0.214%     str = str || y;
 7         );
 8
 9  0.054% str = "";
10
11  0.250% For( i = 0, i <= 9, i++,
12  2.859%     stringFunction(i);
13         );
14
15 11.723% Print( "Finished.", str );
16
```
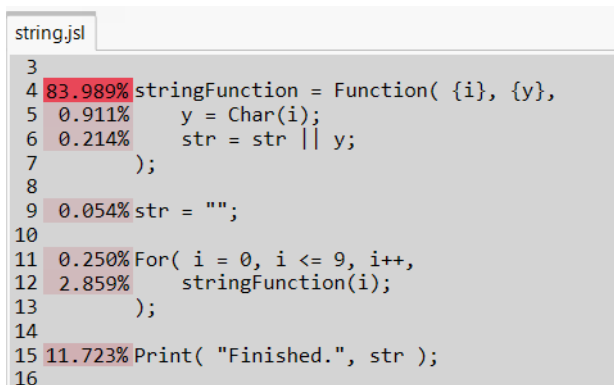
In the left margin, the selected statistics are displayed. Percent of time is displayed by default. Click the **Show Profile by Count** button $\boxed{\Sigma}$ to switch to percent of statement counts instead. The left margin is color-coded to allow for quick identification of problematic performance areas.

JSL Building Blocks

Learning the Basics of JSL

Studying the syntax and basics of JSL is crucial, whether you are a beginning or an advanced user. Some concepts (such as loops and variables) are common among many scripting languages, but punctuation rules differ in JSL.

Figure 5.1 Example of a JSL Script
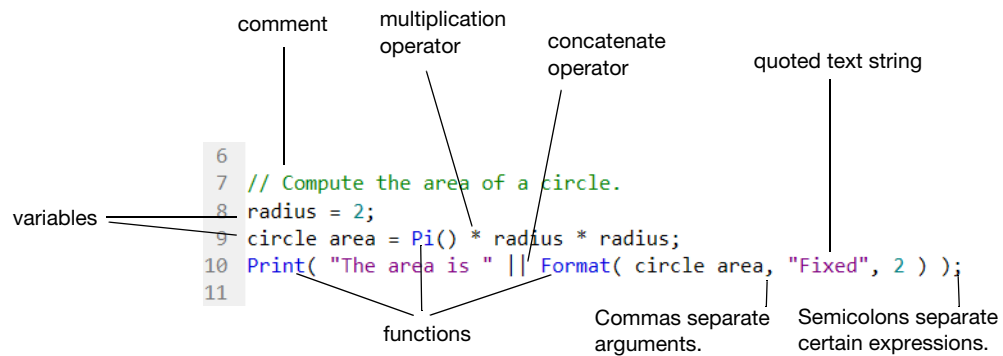
This chapter introduces you to the basic concepts of JSL, from syntax rules and file path conventions to conditions and namespaces.

Contents

JSL Syntax Rules

All scripting and programming languages have their own syntax rules. JSL looks familiar if you have programmed in languages such as C and Java. However, rules for punctuation and spaces differ in JSL.

The following sections describe JSL syntax rules for the following basic components of the language.

Value Separators

Words in JSL are separated by parentheses, commas, semicolons, spaces, and various operators (such as +, –, and so on). This section describes the rules for using these separators and delimiters in JSL.

Commas

A comma separates items, such as items in a list, rows in a matrix, or arguments to a function.

```
my list = {1, 2, 3};
your list = List( 4, 5, 6 ) ;
my matrix = [3 2 1, 0 -1 -2];
If( Y < 20, X = Y );
Table Box( String Col Box( "Age", a ) );
```

Note: To glue a sequence of commands into a single argument inside a function, separate each sequence with a semicolon. See "Semicolons" on page 99.

Parentheses

Parentheses have several purposes in JSL:

- Parentheses group operations in an expression. The following parentheses group the operations in the `If()` expression.

```
y = 10;
If(
    Y < 20, X = Y,
    X = 20
);
```

- Parentheses delimit arguments to a function. In the following example, parentheses enclose the argument to the `Open()` function.

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
```

- Parentheses also mark the end of a function name, even when arguments are not needed. For example, the Pi function has no arguments. However, the parentheses are required so that JMP can identify *Pi* as a function.

  ```
  Pi();
  ```

Note: Be careful that parentheses match. Every (needs a), or errors result.

The script editor can match fences (parentheses, brackets, and braces). Press Ctrl+] (Command+B on macOS) with your cursor in any part of a script. The editor searches for fences, highlighting the text between the first set of opening and closing fences that it finds. Repeat this process to highlight the next-higher fence. See "Match Parentheses, Brackets, and Braces" on page 64 in the "Scripting Tools" chapter for an example.

Semicolons

Expressions separated by a semicolon are evaluated in succession, returning the result of the last expression. In the following code, 0 is assigned to the variable i and then 2 is assigned the variable j.

```
i = 0;
j = 2;
```

You can also use semicolons to join arguments that are separated by commas as shown in the following If() expression.

```
If( x < 5, y = 3; z++; );
```

The semicolon in other languages is used as an expression terminator character. In JSL, the semicolon is a signal to continue because more commands might follow. For more information about separating expressions with semicolons, see "Alternatives for Gluing Expressions Together" on page 119.

Semicolons at the end of a script or at the end of a line of arguments are harmless, so you can also think of semicolons as terminating characters. Trailing semicolons are allowed at the end of a script stream and after a closing parenthesis or closing curly brace. In fact, terminating each complete JSL expression with a semicolon helps avoid errors when you copy and paste small scripts into a larger one.

The semicolon is equivalent to the Glue() function. See "Operators" on page 103 for more information about semicolons and Glue().

Double Quotes

Double quotes enclose text strings. Anything inside double quotes is taken literally, including spaces and upper- or lower-case; nothing is evaluated. If you have **"Pi() ^ 2"** (inside double quotes), it is just a sequence of characters, not a value close to ten.

You do have to be a careful with text strings. Extra spaces and punctuation do affect the output, because text strings inside double quotes are used literally, exactly as you enter them.

To have double quotes inside a quoted string, precede each quotation mark with the escape sequence \! (backslash-bang). For example, run the following script and look at the title of the window:

```
New Window( "\!"Hello\!" is a quoted string.",
   Text Box( Char( Repeat( "*", 70 ) ) )
);
```

Table 5.1 Escape Sequences for Quoted Strings

\!b	blank
\!t	tab
\!r	carriage return only
\!n	linefeed (newline) only
\!N	inserts line breaking characters appropriate for the host environment[a]
\!f	formfeed (page break)
\!0	null character
	Note: The null character is dangerous to use, because it is typically treated as the end of the string. Be sure to type the number zero, not the letter O.
\!\	backslash
\!"	double quotation mark

a. On macOS, this escape sequence is CR (carriage return character, hexadecimal '0D'). On Windows, this sequence is CR LF (carriage return followed by a linefeed, hexadecimal '0D0A').

Sometimes, long passages require a lot of escaped characters. In these cases, use the notation \[...]\ and everything between the brackets does not need or support escape sequences. Here is an example where \[...]\ is used inside a double-quoted string.

```
jslPhrase = "The JSL to do this is :\[
   a = "hello";
   b = a|| " world.";
   Show(b);
   ]\ and you use the Submit command to run it.";
```

Spaces

JSL allows whitespace characters inside names; spaces, tabs, returns, and blank lines inside or between JSL words are ignored. This is because most JSL words come from the user interface, and most of those commands have spaces in them. For example, the JSL expression for the Fit Model platform is `Fit Model()` or `FitModel()`.

Spaces inside an operator or between digits in a single number are not allowed. In these cases, the script generates errors. For example, you cannot put spaces between the two plus signs in `i++ (i+ +)` or in numbers (`4  3` is not the same as `43`).

Note: Why does JSL allow whitespace characters inside names? For one reason, the names of commands and options match the equivalent commands in JMP menus and windows. Another reason is that data table column names often include spaces.

Numbers

Numbers can be written as integers, decimal numbers, dates, times, or date-time values. They can also be included in scientific notations with an `E` preceding the power of ten. For example, these are all numbers:

    .   1   12   1.234 3E3 0.314159265E+1 1E-20 01JAN98

Note: A single period by itself is considered a missing numeric value (sometimes called *NaN* for "not a number").

For more information about dates, times, and date-time values, see "Date-Time Functions and Formats" on page 150 in the "Types of Data" chapter. See "Currency" on page 164 in the "Types of Data" chapter for more information about combining numbers with currency symbols.

Names

A name is simply something to call an item. When you assign the numeric value 3 to a variable in the expression `a = 3`, `a` is a name.

Commands and functions have names, too. In the expression `Log( 4 )`, `Log` is the name of the logarithmic function.

Names have a few rules:

- Names *must* start with an alphabetic character or underscore and can continue with the following:
  - alphabetic characters (a-z A-Z)

- numeric digits (0-9)

- whitespace characters (spaces, tabs, line endings, and page endings)

- mathematical symbols in Unicode (such as the Greek small alpha letter α)

- a few punctuation marks or special characters (apostrophes ('), percent signs (%), periods (.), backslashes (\), and underscores (_))

- When comparing names, JMP ignores whitespace characters (such as spaces, tabs, and line endings). Upper case and lower case characters are not distinguished. For example, the names `Forage` and `for age` are equivalent, despite the differences in white space and case.

You can still have a name that is any other sequence of characters. If the name does not follow the rules above, it needs to be quoted and `followed by "n"`. For example, to use a global variable with the name `taxable income(2011)`, you must use `"taxable income(2011)"n` every time the variable appears in a script:

```
"taxable income( 2011 )"n = 456000;
tax = .25;
Print( tax * "taxable income( 2011 )"n) ;
    114000
```

Note: Previously, the `Name()` parser directive wrapped the column name that contained illegal characters. `Name()` is deprecated.

For more information about how JMP interprets names, see "Rules for Name Resolution" on page 111.

Comments

Comments are notes in the code that are ignored by the JSL processor (or *parser*). You include comments to describe sections of the script. Comments are also convenient for removing portions of a script temporarily. For example, you can insert comment symbols around code that might be causing an error and then rerun the script.

Type the comment symbols around code that you want to comment. The following example shows code commented with /* */ in the middle of a line. When the script is run, JMP considers both expressions to be identical.

```
tax /*percentage*/ = .25;
tax = .25;
```

Table 5.2 describes the comment symbols.

Table 5.2 Comment Symbols

Symbol	Syntax	Explanation
//	// comment	Begins a comment. The comment does not have to be at the beginning of a line, but everything up to the end of the line is a comment.
/* */	/* comment */	Begins and ends a comment. This comment can appear in the middle of a line. Script text before and after the comment is parsed normally.
//!	//!	Add //! to the first line of the script, and the script runs automatically when opened in JMP. (In other words, the script editor does not open.)

Operators

An operator is a one- or two-character symbol used to perform common arithmetic actions, compare values, construct lists, subscript into a data element, send messages, concatenate lists, scope names, glue expressions, and end expressions.

Arithmetic operators come in several varieties:

- *infix* (with arguments on either side, such as + in 3 + 4, or = in a = 7)
- *prefix* (with one argument on its right side, such as !a for logical negation)
- or *postfix* (with one argument on its left side, such as a++ for incrementing a

To make writing expressions easier, JSL uses certain special character operators that are alternatives to using functions. These operators have the same meaning as if the phrase had been written as a function. For example, the following two expressions are equivalent.

```
Net Income After Taxes = Net Income – Taxes;
Assign( Net Income After Taxes, Subtract( Net Income, Taxes ) );
```

The assignment operation can be written either as the Assign() function or as an infix operator =. Similarly, subtraction can be done with the Subtract() function or the infix minus sign; they are equivalent inside JMP.

Note: Usually white space is ignored in JSL, so that "netincomeaftertaxes" and "net income after taxes" are the same thing. There is one exception: the two-character operators must *not* have a space between characters. Always enter the following operators without spaces in between:

```
||, |/, <=, >=, !=, ==, +=, -=, *=, /=,  ++, --, <<, ::, :*, :/, /*, */
```

Another common operator is the semicolon (;). You use the semicolon to:

- Separate yet join one expression to another in a programming sequence. The semicolon returns the result of its last argument, so `a;b` is the same as `Glue(a,b)`.

- End an expression. Though the semicolon is permitted at the end of an expression, it is not the expression terminator used in other languages.

An expression can contain more than one operator. In these instances, the operators are grouped in order of precedence with decreasing priority. For example, * takes precedence over +:

```
a + b * c
```

So `b * c` is evaluated first, and then the result is added to `a`.

+ takes precedence over –:

```
a + b * c - d
```

So `b * c` is evaluated, and then the result is added to `a`. `d` is then subtracted from the result of `a + b * c`.

Table 5.3 shows operators shaded in order of precedence and each operator's function equivalent.

Table 5.3 Operators and Their Function Equivalents in Order of Precedence

Operator		Function Syntax	Explanation
`{ }`	List	*{a,b}* `List(a,b)`	Construct a list.
`[ ]`	Subscript	*a[b,c]* `Subscript(a,b,c)`	Subscripts identify specific elements within a data element *a*, where *a* could be a list, a matrix, a data column, a platform object, a display box, and so on.

Table 5.3 Operators and Their Function Equivalents in Order of Precedence *(Continued)*

Operator		Function Syntax	Explanation
++	Post Increment	`a++` `Post Increment(a)`	Adds one (1) to *a*, in place. JMP does not have a pre-increment operator. Instead, use the `Add To()` operator, which is (+=).
--	Post Decrement	`a--` `Post Decrement(a)`	Subtracts one (1) from *a*, in place. JMP does not have a pre-decrement operator. Instead, use the `Subtract To()` operator, which is (-=).

^	Power	`a^b`
		`Power(a,b)`
		`Power(x)`
			Raise *a* to exponent power *b*. With only one argument, 2 is assumed as the power, so `Power(x)` computes x^2.

−	Minus	`-a` `Minus(a)`	Reverses sign of *a*.
!	Not	`!a` `Not(a)`	Logical Not. Maps nonzero (or true) values to 0 (which means false). Maps 0 (or false) values to 1 (which means true).
*	Multiply	`a*b` `Multiply(a,b)`	Multiplies *a* by *b*.
:*	EMult	`a:*b` `EMult(a,b)`	Elementwise multiplication for matrices *a* and *b.* (Each element in matrix *a* is multiplied by each element in matrix *b*.)

/	Divide	`a/b`
		`Divide(a,b)`
		`Divide(x)`
			`Divide(a, b)` divides *a* by *b*.
			
			`Divide(x)` interprets the argument as a denominator and implies 1 as the numerator, yielding the reciprocal 1/*x*.

:/	EDiv	`a:/b` `EDiv(a,b)`	Elementwise division for matrices *a* and *b*. (Each element in matrix *a* is divided by each element in matrix *b*.)

Table 5.3 Operators and Their Function Equivalents in Order of Precedence *(Continued)*

Operator		Function Syntax	Explanation
+	Add	*a*+*b* Add(*a*,*b*)	Adds *a* and *b*.
–	Subtract	*a*–*b* Subtract(*a*,*b*)	Subtracts *b* from *a*.
||	Concat	*a*||*b* Concat(*a*,*b*)	Joins two or more strings; two or more lists; and horizontally concatenates matrices. See "Concatenate Lists" on page 200 in the "Data Structures" chapter or the *JSL Syntax Reference*.
|/	VConcat	*matrix1*|/*matrix2* VConcat(*matrix1*, *matrix2*)	Vertically concatenate matrices. (Use || or Concat() to horizontally concatenate matrices.)
::	Index	*a*::*b* Index(*a*,*b*)	For matrices, generates the integers from *a* to *b*. (Colons are also used as prefix operators for scoping, where :a means data table column *a*, and ::a means JSL global variable *a*. See "Scoping Operators" on page 114.)
<<	Send	object << message Send(*object*, *message*)	Send *message* to *object*.
==	Equal	*a*==*b* Equal(*a*,*b*)...	Boolean values for comparisons. They all return 1 if true, 0 if false. Missing values in either *a* or *b* causes a return value of missing, which evaluates as neither true nor false. See "Missing Values" on page 135, for treatment of missing values.
!=	Not Equal	*a*!=*b* Not Equal(*a*,*b*)...	
<	Less	*a*<*b* Less(*a*,*b*)...	
<=	Less or Equal	*a*<=*b* Less or Equal(*a*,*b*)	
>	Greater	*a*>*b* Greater(*a*,*b*)	
>=	Greater or Equal	*a*>=*b* Greater or Equal(*a*,*b*)	

Table 5.3 Operators and Their Function Equivalents in Order of Precedence *(Continued)*

Operator		Function Syntax	Explanation
<=, <	Less Equal Less	`a<=b<c` `Less Equal Less(a,b,c)`	Range check. Return 1 if true, 0 if false. Missing values in either *a* or *b* propagate missing values.
<, <=	Less Less Equal	`a<b<=c` `Less Less Equal(a,b,c)`	
&	And	`a&b` `And(a,b)`	Logical And. Returns true if both are true. If the value on the left is false, the value on the right is not evaluated. See "Missing Values" on page 135, for treatment of missing values.
|	Or	`a|b` `Or(a,b)`	Logical Or. Returns true if either or both are true. See "Missing Values" on page 135, for treatment of missing values.
=	Assign	`a=b` `Assign(a,b)`	Put the value of *b* into *a*. Replaces the current value of *a*.
+=	Add To	`a+=b` `AddTo(a,b)`	Add the value of *b* into *a*.
-=	Subtract To	`a-=b` `SubtractTo(a,b)`	Subtract *b* from *a*, and put back into *a*.
*=	Multiply To	`a*=b` `MultiplyTo(a,b)`	Multiply *b* with *a*, and put back into *a*.
/=	Divide To	`a/=b` `DivideTo(a,b)`	Divide *b* into *a*, and put back into *a*.
;	Glue	`a;b` `Glue(expr, expr, ...)`	First do *a*, and then do *b*.

Boolean Values

A Boolean value enables or disables an option. To enable the option, use 1, `true`, or `yes` as the argument.

```
obj << Unequal Variances( 1 );
obj << Unequal Variances( "true" );
```

```
obj << Unequal Variances( "yes" );
```

Note: Quoting the value prevents it from being evaluated as a variable.

Omitting the argument also enables the option. This is the default for all but row state messages.

```
obj << Unequal Variances;
obj << Unequal Variances();
```

To disable the option, use 0, false, or no as the argument.

To toggle an option (turn off an enabled option or turn on a disabled option), use "toggle".

Notes:

- The quoted "present", "absent", "on", "off", "switch", and "flip" are supported for backwards compatibility.

- For row state messages, an empty argument always toggles the option.

Global and Local Variables

Variables are names that hold values, which you reference later in scripts. There are two types of variables:

- *Global variables* are shared among all scripts that you run in a JMP session.

- *Local variables* apply only to the script context in which you define them. They can also be local to only a piece of a script, as with variables local to a particular function.

To limit the scope of variables, you can define them in a *namespace*, which is a collection of variables, functions, and other unique names. JMP has a single global variable namespace that all scripts use by default. When you use a name plainly, without a qualifying syntax, the name is an *unscoped* variable and therefore in the global namespace.

```
x = 1;
```

Local Namespaces

Putting variables in the global namespace can cause conflicts. When two scripts have variables with the same names, the value of the variable in the last script that you run last overwrites the variable's value in the first script.

To prevent this problem, we recommend that you begin each script with the following line:

```
Names Default To Here( 1 );
```

The `Names Default To Here( 1 );` function makes all unscoped variables in the script local to that script and does not affect the global variable namespace. See "Advanced Scoping and Namespaces" on page 272.

Notes:

• The `Names Default to Here` setting is true by default for custom menus and toolbar buttons. A script that runs when you select a custom menu item or click a custom toolbar button does not affect global variables.

• The `Names Default to Here` setting is false by default for table scripts. If a table script specifies `Names Default to Here(1)`, this setting lasts only as long as the table script runs. After the script finishes running, the setting is reset to whatever it was before the script ran.

Named Namespaces

You can also create a variable in a specific namespace. In the following example, the x variable is created in the aa namespace:

```
aa:x = 1;
```

Preceding local variables with the `Local()` function is another option. Both a and b are local variables in the following expression:

```
Local( {a = 1, b}, ... );
```

Scoping operators also distinguish a global variable from a local variable. See "Rules for Name Resolution" on page 111.

The following sections describe functions that help you manage variables.

Show Symbols, Clear Symbols, and Delete Symbols

The `Show Symbols()` function lists all variables and namespaces that are defined both globally and in the local script, along with their current values. Here is an example of `Show Symbols()` messages that are shown in the log:

```
Show Symbols();
    // Here
    a = 5;
    b = 6;
    // 2 Here

    // Global
    c = 10;
    // 1 Global
```

Tip: The JSL debugger also shows you the values of variables and namespaces. See "Debug or Profile Scripts" on page 77 in the "Scripting Tools" chapter.

The `Clear Symbols()` function erases the values set for variables that are defined both globally and in the local script. For example, after you clear and then show symbols, the variables are empty.

```
Clear Symbols();
Show Symbols();
   // Here
   a = Empty;
   b = Empty;
   // 2 Here

   // Global
   c = Empty;
   // 1 Global
```

Note: The older `Show Globals()` and `Clear Globals()` functions are aliases of the newer `Show Symbols()` and `Clear Symbols()` functions.

To remove *all* global variables and namespaces, use the function `Delete Symbols()`. After the last `Show Symbols()` in the following script is run, nothing shows up in the log. All variables have been completely removed from memory.

```
Delete Symbols();
Show Symbols();
```

To list variables in all namespaces, use `Show Namespaces()`. To delete only a specific namespace, use `ns << Delete`. `Clear Symbols()` and `Delete Symbols()` do not clear or delete variables in each namespace, although they do clear and delete variables that contain references to namespaces. See "Rules for Name Resolution" on page 111 for more information about unscoped variables.

Note: `Clear Symbols()` and `Delete Symbols()` break all scripts that are currently in use. These functions can be very useful in a programming and debugging environment, but do not include them in any script that you plan to distribute. If you include `Names Default To Here(1)` in your scripts, clearing and deleting global symbols is unnecessary.

Lock and Unlock Symbols

If you want to lock a variable to prevent it from being changed, use the `Lock Symbols()` function. (`Lock Globals()` is an alias.)

```
Lock Symbols( name1, name2, ... );
```

To release the lock and enable the global to be changed, use the `Unlock Symbols()` function. (`Unlock Globals()` is an alias.)

```
Unlock Symbols( name1, name2, ... );
```

The primary use of these two commands is to prevent inadvertent changes to variables. For example, locking a variable prevents `Clear Symbols`() from clearing a variable that is being used by another script.

Note: You cannot use `Lock Symbols()` to lock a namespace. Instead, use `ns <<Lock Namespace.`

Hide a Global Variable

To hide (or protect) a global variable, put two underscores (__) before the name. Protecting a global variable means that it will be hidden and can not be examined or shown. However, the behavior is different depending upon the context.

This example will not print the variable because it is protected:

```
Show(::__xyz);
```

This example returns a NULL value for the protected variable:

```
::a = Name Expr( ::__xyz );
```

This example does return a value for the protected variable:

```
Show( Eval( ::__xyz ) );
```

Rules for Name Resolution

In JMP, you identify the following types of objects by a name:

- Columns and table variables in a data table
- Global variables, which hold values for the remainder of a JMP session
- Scriptable object types
- Arguments and local variables inside formulas

Most of the time, you can just use an object's name directly to refer to the object. Consider the following example:

```
ratio = height / weight;
```

Depending on the complexity of your script, it might be obvious that `ratio` is a variable and `height` and `weight` are data table column names. But what if the meanings are ambiguous? A script might use `ratio` as a global variable and as column names.

Variable Names Used as Arguments

Message arguments are evaluated as a rule. In previous versions of JMP, some messages interpreted a variable name as a string value. For example, the following script results in a linear scale in the current version of JMP and a log scale in JMP 11.

```
Log = "Linear";
axis box << Scale( Log );
```

The syntax `Scale( "Log" )` is preferred for setting a string literal. To catch ambiguous instances, set the "Allow Unquoted Strings in JSL" General preference to "Yes (with a warning)" or "No".

Rules for Resolving Names

JMP interprets object names using *name resolution*. The following rules are applied sequentially to unscoped names:

1. If the variable is followed by a pair of parentheses `( )`, look it up as a function.

2. If the variable is prefixed by `:` scope operator or an explicit data table reference, look it up as a data table column or table variable.

3. If the variable is prefixed by `::` scope operator, look it up as a global variable.

4. If the variable is an explicit scope reference (such as `group:vowel`), look it up in the user-defined `group` namespace.

5. If the variable is in a `Local` or `Parameter` function, look it up as a local variable. If it is nested, repeat until a function call boundary is found.

6. If the variable is in a user-defined function or method, look it up as a function or method argument or local variable.

7. Look the variable up in the current scope and its parent scope. Repeat until the `Here` scope is encountered.

8. Look the variable up as a variable in the `Here` scope.

9. Look the variable up as a global variable.

10. If `Names Default to Here(1)` is at the top of the script, stop looking. The scope is local.

11. Look the variable up as a data table column or table variable. Columns are preferred over table variables.

12. Look the variable up as an operator or a platform launch name (for example, Distribution, Bivariate, Chart, and so on).

13. When the name cannot be found:

   – If the name is used in a reference, print and error to the log.

   – If the name is used as the target of an assignment (as an L-value), test the following:

      If the variable is preceded by `::` scope operator, create and use a global variable.

      If the variable is an explicit scope reference, create and use the variable in the specified namespace or scope.

      If `Names Default to Here(0)` is at the top of the script, create a global variable.

      If `Names Default to Here(1)` is at the top of the script, create a `Here` namespace variable.

Exceptions

• Some names are variables that refer to an object such as a data table, data column, or platform; they are not used for getting or setting a value. These names are passed through (or interpreted literally) rather than resolved.

• For function definitions, column formulas, and Nonlinear platform formulas, the scope is the same for each row in a column.

• If a name is a direct reference to a column in a data table that has been closed, the name is resolved again to that column when the table is reopened.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show( :weight << Get As Matrix ); // weight  resolves to a column name
Close( dt, NoSave );
Show( :weight << Get As Matrix ); // weight  cannot be resolved
dt = Open( "$SAMPLE_DATA/Big Class.jmp" ); // reopen data table
Show( :weight << Get As Matrix ); // weight  resolves to a column name
```

However, the following example does not resolve the variable to the second instance of the data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( dt, 5); // col is Column( "weight" );
Close( dt, NoSave );
dt = Open( "$SAMPLE_DATA/Big Class.jmp" ); // reopen data table
Show( col << Get As Matrix ); // the reference to the first data table no
    longer exists
```

The following sections describe how JMP resolves the names of data table columns. For more information about name resolution, see "Advanced Scoping and Namespaces" on page 272 in the "Programming Methods" chapter.

Scoping Operators

Using *scoping operators* is an easy way to help JMP resolve ambiguous names (for example, when a name refers to both a variable and a column name).

In the following example, the prefix double-colon operator (`::`) identifies z as a global variable. The single-colon prefix operator (`:`) identifies x and y as column names.

```
::z = :x + :y;
```

Tip: The `Names Default to Here(1)` function also affects name resolution. See "Names Default To Here and Global Variables" on page 274 in the "Programming Methods" chapter.

Two JSL functions are interchangeable with scoping operators. Table 5.4 describes the functions and syntax.

Table 5.4 Scoping Operators

Operator and Equivalent Function	Function Syntax	Explanation

`:` `As Column`	`:name`
	`dt:name`
	`As Column(dt, name)`
		Forces *name* to be evaluated as a data table column. The optional data table reference argument, *dt*, sets the current data table. See "Scoped Column Names" on page 114 for examples.

`::` `As Global`	`::name` `As Global(name)`	Forces *name* to be evaluated as a global variable. **Note:** The double-colon is also used as an infix operator to represent ranges.

Scoped Column Names

Scoping column names is the simplest way to prevent conflicts with variable names. Use scoping operators to force names in a script to refer to columns.

1. The prefix colon (`:`) means that the name refers to a table column or table variable only, never a global variable. The prefix colon refers to the current data table context.

   ```
   :age;
   ```

2. The infix colon (`:`) operator extends this notion by using a data table reference to specify which data table has the column. This is particularly important when multiple data tables are referenced in a script.

In the following example, the dt variable sets the data table reference to Big Class.jmp. The infix colon separates the data table reference and age column.

```
dt = Data Table( "Big Class.jmp" );
dt:age // The colon is an infix operator.
```

As Column() achieves the same results:

```
dt = Data Table( "Big Class.jmp" );
As Column( dt, age );
```

Therefore, the following expressions are equivalent when only Big Class.jmp is open:

```
:age;
As Column( dt, age );
dt:age;
```

The Column function can also identify a column. For Big Class.jmp, the following expressions all refer to age, the second column in the table:

```
Column( "age" );
Column( 2 );
Column( dt, 2 );
Column( dt, "age" );
```

Prevent Column Name and Variable Name Conflicts

To avoid conflicts, use unique column and variable names or scope the names.

• When a global variable and a column have the same name, the global variable name takes precedence. In this situation, you must scope the column name.

```
::age = [];
age = :age << Get As Matrix;
```

• To avoid ambiguity between the global variable and column name, scope both variables.

```
::age = :age << Get As Matrix;
```

• If more than one data table might be open, assign data table references to variables. Scope your columns to the appropriate table.

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/Students.jmp" );
::age = dt1:age << Get As Matrix;
::height = dt2:height << Get As Matrix;
```

Note that JMP evaluates column formulas through each consecutive cell of the column, so scoping the column name is usually unnecessary. However, if a variable assigned in a formula has the same name as a column, you must scope the column name. See "Scoped Names" on page 276 in the "Programming Methods" chapter.

Tip: The Names Default to Here(1) function also affects name resolution. See "Names Default To Here and Global Variables" on page 274 in the "Programming Methods" chapter.

Unscoped Column Names

Sometimes an unscoped name gets or sets a value. JMP resolves it as a column in a data table (rather than a global variable) under these circumstances:

- if no global variable, local variable, or an argument using that name already exists,
- and the data table in context has a column of that name,
- and
  - either the current row is set to a positive value
  - or the name is subscripted (for example, the subscript [1] in weight[1] selects the first value in the weight column).

Exception

In column formulas and Nonlinear formulas, column names take precedence over global variables.

Set the Current Data Table Row

By default, the current row is 0, an illegal row number. So the following expression assigns a missing value to the *ratio* global variable:

```
ratio = height / weight;
```

Specify the row number with the Row() function. In the following example, the row is set to 3. The height in that row is divided by the weight, and the result is assigned to the *ratio* global variable.

```
Row() = 3;
ratio = height / weight;
```

Another possibility is to use subscripts to specify the row number. The following expression divides the height in row 3 by the weight in row 4.

```
ratio = height[3] / weight[4];
```

Specifying the row number is unnecessary when the script iterates a row at a time down the whole column. The following example creates the ratio column. For each row, the height is divided by the weight.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "ratio" );
For Each Row( :ratio = height / weight );
```

JMP evaluates formulas and calculates pre-evaluated statistics iteratively down a column. In these instances, identifying the row number is also unnecessary. (Pre-evaluated statistics are single numbers computed from the data table, as discussed in "Pre-Evaluated Statistics" on page 466 in the "Data Tables" chapter.)

Name Resolution in Automatically Run Scripts

In an `Open()` function, you can specify that the script run automatically when it's opened.

```
Open( "$MY_SCRIPTS/A.jsl", Run JSL( 1 ) );
Open( "$MY_SCRIPTS/B.jsl", Run JSL( 1 ) );
```

You can also type `//!` on the first line of the script to run the script automatically.

In these instances, the only shared names between `A.jsl` and `B.jsl` are globals and namespaces (not here and not local).

To apply the names in `A.jsl` to `B.jsl`, include `B.jsl` in `A.jsl` instead of opening and running both scripts.

```
Include( "$MY_SCRIPTS/B.jsl" );
```

Troubleshoot Variables and Column Names

When you reference a column name using `As Name()`, and `Names Default To Here( 1 )` is set, JMP returns a variable reference. That reference is then processed using the standard reference rules.

In the following example, there is no `height` variable in the `Here:` scope, so JMP returns an error.

```
Names Default To Here( 1 );
Open( "$SAMPLE_DATA/Big Class.jmp" );
As Name( "height" )[3]; // converts height to a variable reference
   As Name( "height" )[/*###*/3];
```

The `/*###*/` characters in the log indicate the location of an error in the script execution.

To prevent this problem, use one of the following methods:

• Use `As Column()` instead of `As Name()`:

```
Names Default To Here( 1 );
Open( "$SAMPLE_DATA/Big Class.jmp" );
As Column( "height" )[3]; // converts height to a data column reference
```

• Explicitly scope `height` with `As Name()`:

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:( As Name( "height" ) )[3]; // scopes height as a data column reference
```

These scripts return 55, the value of height in the third row of Big Class.jmp.

Troubleshoot Variables and Keywords

Name resolution errors can also occur when a variable and unquoted keyword have the same name. For example, one argument for <<Preselect Role() is "Y". Quote this argument if your script also uses Y as a variable.

Frequently Asked Questions about Name Resolution

Should you always scope?

Yes. When in doubt, scope. Scoping is especially important in scripts that might be used by many people on a variety of data tables; you will not necessarily know whether a name is used in two contexts (such as for both a global variable and column name).

If you are writing such scripts, consider using explicit scoping and namespaces. See "Advanced Scoping and Namespaces" on page 272 in the "Programming Methods" chapter.

Prefix scope operators do not take run-time overhead after the first resolution. Infix scope operators, which follow data table references, always take run-time overhead.

What is the difference between a column reference and a column referred to by name? If I have a column reference in a global variable, how do I assign a value to a cell in the column?

With a column reference, you can send messages to change specific characteristics of the column or to access its values (for example, coloring cells or setting a formula).

When a column has been assigned to a global variable, assign a value to a cell in the column using a subscript. Suppose that the name of the column height has been assigned to the x variable:

```
x = Column( "height" );
```

Assign a value to the third row in the height column:

```
x[3] = 64 // sets the third row of height to 64
```

Note: The current row in a JSL script is *not* determined by selecting rows or positioning your cursor in the row. The current row is defined to be zero (no row) by default. You can set a current row with Row() (for example, Row() = 3). Please note that such a setting only lasts for the duration of that script. Then, Row() reverts to its default value, zero. This behavior means that submitting a script all at once can produce different results than submitting a script a few lines at a time.

Another way to establish a current row for a script is to enclose it in For Each Row(). This method executes the script once for each row of the current data table. For an example, see "If" on page 126. See the "Data Tables" chapter on page 321 for more information about working with data tables.

Which Has Precedence When Scoping, ":" or "[ ]"?

Scoping occurs before subscripting. This means that these two lines of code are equal:

```
dataTable:colName[i]
(dataTable:colName)[i]
```

Alternatives for Gluing Expressions Together

You can separate expressions with a semicolon, either on the same line or on different lines. JMP then evaluates each expressions in succession, returning the result of the last one. Here is an expression that first sets a to 2 and then sets b to 3:

```
a = 2;
b = 3;
```

The semicolon joins the two expressions and returns the value of the last one. So if x = (a = 2; b = 3), the value of x is 3.

The Glue() function returns the result of the last expression. This function is equivalent to using semicolons. The following expressions both return 3:

```
Glue( a = 2, b = 3 );
a = 2; b = 3;
```

The First() function also evaluates each argument sequentially but returns the result of the first expression. The following expression returns 2:

```
First( a = 2, b = 3);
```

Example

What does First() do in the following script?

```
x = 1000;
First( x, x = 2000 );
```

The First() function returns the value of x (1000). 2000 is then assigned to x.

Iterate

JSL provides the For(), While(), Summation(), and Product() functions to repeat (or *iterate*) actions according to the conditions that you specify.

Note: A similar function called For Each Row() is for iterating actions over rows of a data table. See "If" on page 126 for an example. "Additional Ways to Access Data Values" on page 459 in the "Data Tables" chapter also describes iterating through table rows.

- "For"
- "While"
- "Summation"
- "Product"
- "Break and Continue"

For

The For() function expects four arguments separated by commas. The first three arguments are rules for how many times to repeat the loop, and the fourth is what to do each time the loop is executed.

Here is the basic syntax for For():

```
For( initialization, while, iteration, body );
```

For example, the following script sums the numbers from 0 to 20:

```
s = 0;
For( i = 0, i <= 20, i++, s += i );
```

The script works like this:

`s = 0;`	Sets the s variable to 0. This variable holds the sum.
`For(`	Begins the For() loop.
`    i = 0,`	Sets the initialization variable (i) to 0. This expression is performed only once.

`i <= 20,`	Each time the loop begins, compares `i` to 20. As long as `i` is less than or equal to 20, continue evaluating the loop. If `i` is greater than 20, immediately break out of the loop.
`i++,`	At the end of the loop, increments `i` by 1. Note that this step is done after the body of the loop (next line) is evaluated.
`s += i`	Evaluates the body of the loop. Adds the value of `i` to `s`. After the body is finished, *i* is incremented (previous line).
`);`	Ends the loop.

Infinite Loops

`For` loops that always evaluate as true create an infinite loop, which never stops. To stop the script, press Esc on Windows (or Command+period on macOS). You can also select **Edit > Stop Script**. On macOS, **Edit > Stop Script** is available only when the script is running.

Comparing For Loops in JSL to C and C++

The JSL `For()` loop works just like it does in the C (and C++) programming language, although the punctuation is different.

Tip: If you know C, watch out for the difference between JSL and C in the use of semicolons and commas. In JSL, `For()` is a function where commas separate arguments and semicolons join expressions. In C, `for` is a special clause where semicolons separate arguments and commas join them.

While

A related function is `While()`, which repeatedly tests the condition and evaluates its body script as long as the condition is true. The syntax is:

```
While( condition, body );
```

For example, here are two different programs that use a `While()` loop to find the least power of 2 that is greater than or equal to `x` (287). The result of both programs is 512.

```
x = 287;

// loop 1:
y = 1;
While( y < x, y *= 2 );
Show( y );
```

```
// loop 2:
k = 0;
While( 2 ^ k < x, k++ );
Show( 2 ^ k );
```

The scripts work like this:

`x = 287;`	Sets x to 287.
`// loop 1`	
`y = 1;`	Sets y to 1.
`While(`	Begins the `While()` loop.
`y < x,`	As long as y is less than x, continues evaluating the loop.
`y *= 2`	Multiplies 1 by 2 and then assigns the result to y. The loop then repeats while y is less than 287.
`);`	Ends the loop.
`Show(y);`	Shows the value of y (512).
`// loop 2`	
`k = 0;`	Sets k to 0.
`While(`	Begins the `While()` loop.
`2 ^ k < x,`	Raises 2 to the exponent power of k and continues evaluating the loop as long as the result is less than 287.
`k++`	Increments k to 1. The loop then repeats while 2 ^ k is less than 287.
`);`	Ends the loop.
`Show(2 ^ k);`	Shows the value of 2 ^ k (512).

As with `For()` loops, `While()` loops that always evaluate as true create an infinite loop, which never stops. To stop the script, press Esc on Windows (or Command-period on macOS). You can also select **Edit > Stop Script**. On macOS, **Edit > Stop Script** is available only when the script is running.

Summation

The `Summation()` function adds the body results over all `i` values. The syntax is:

```
Summation( initialization, limitvalue, body );
```

For example:

```
s = Summation( i = 1, 10, i );
```

returns 55, the result of 1+2+3+4+5+6+7+8+9+10.

The script works like this:

`s =`	Sets the `s` variable to the value of the function.
`Summation(`	Begins the `Summation()` loop.
`i = 1,`	Sets `i` to 1.
`10,`	Sets the limit of `i` to 10.
`i`	All values of `i` from 1 to 10 are added together, resulting in 55.
`);`	Ends the loop.

This behavior is similar to Σ in the Formula Editor. The following expression:

```
Summation( i = 1, N Row(), x ^ 2 );
```

is equivalent to the following formula in the Formula Editor:

$$\sum_{i=1}^{\text{NRow}} x^2$$

Product

The `Product()` function is similar to `Summation()` except that it multiplies the body results rather than adding them. The syntax is the same as for `Summation()`. For example:

```
p = Product( i = 1, 5 , i );
```

returns 120, the result of 1*2*3*4*5.

In this example, the initial value of `i` is 1, the upper limit is 5, then all integer values of `i` up to 5 are multiplied.

Here is the equivalent in the Formula Editor:

$$\prod_{i\,=\,1}^{5} i$$

Break and Continue

The Break() and Continue() functions give you more control over looping. Break()
immediately stops the loop and proceeds to the next expression that follows the loop.
Continue() is a gentler form of Break(). It immediately stops the current iteration of the loop
and continues with the next iteration of the loop.

Break

Break() is typically used inside a conditional expression. For example:

```
For( i = 1, i <= 5, i++,
   If( i == 3, Break() );
   Print( "i=" || Char( i ) );
);
```

results in:

```
"i=1"
"i=2"
```

The script works like this:

For(	Begins the For() loop.
i = 1,	Sets i to 1.
i <= 5,	As long as i is less than or equal to 5, continues evaluating the loop.
i++,	Increments i by 1. Note that this step is done after the If loop is evaluated.
If(	Begins the If() loop.
i == 3, Break()	If i is equal to 3, breaks the loop.
);	Ends the loop.
Print(	When i is not equal to 3, opens the Print() loop.
"i="	Prints the string "i=" to the log.
||	Places "i=" on the same line as the value that follows.

`Char(i));`	Prints the value of i to the log.
	The `For()` loop then repeats until the value of i is less than or equal to 5, breaking when i equals 3, and breaking and printing only when i is less than 3.
`);`	Ends the loop.

Note that when the `If()` and `Break()` expressions follow `Print()`, the script prints the values of i from 1 to 3, because the loop breaks after "i=3" is printed.

```
"i=1"
"i=2"
"i=3"
```

Continue

As with `Break()`, `Continue()` is typically used inside a conditional expression. For example:

```
For( i = 1, i <= 5, i++,
    If( i < 3, Continue() );
    Print( "i=" || Char( i ) );
);
```

results in:

```
"i=3"
"i=4"
"i=5"
```

The script works like this:

`For(`	Begins the `For()` loop.
`i = 1,`	Sets i to 1.
`i <= 5,`	Evaluates 1 as less than or equal to 5.
`i++,`	Increments i by 1. Note that this step is done after the `If` loop is evaluated.
`If(`	Begins the `If()` loop.
`i < 3, Continue()`	Evaluates i as 1 and continues as long as i is less than 3.
`);`	Ends the `If()` loop.

`Print(`	When i is no longer less than 3, opens the `Print()` loop.
`"i="`	Prints the string `"i="` to the log.
`||`	Places `"i="` on the same line as the value that follows.
`Char(i));`	Prints the value of i to the log.
	The `For()` loop then repeats until the value of i is less than or equal to 5, continuing and printing only when i is equal to or greater than 3.
`);`	Ends the loop.

Conditional Functions

JSL provides five functions to evaluate an expression conditionally:

- "If"
- "Match"
- "Choose"
- "Interpolate"
- "Step"

If

The `If()` function evaluates the first result expression when its condition evaluates as true (a nonzero or nonmissing value). Otherwise, it evaluates the second result expression.

The syntax is:

```
If ( condition, result1, result2 );
```

For example, the following script returns "Young" when the age is less than 12. Otherwise, the script returns "Young at Heart".

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "age group", "Character" );
For Each Row(
    :age group = If( :age <= 12,
        "Young",
        "Young at Heart"
    )
);
```

You can also string together multiple conditions and results. The syntax is:

```
If( condition1, result1,
   condition2, result2,
   ...,
   resultElse );
```

In the preceding example, if condition1 is not true, the function continues evaluating until it finds a true condition. Then that condition's result is returned.

The last result is returned when all conditions are false. And when a value is missing, the missing value is returned. For these reasons, it's very important to include a default result at the end of the expression. Consider the following example, which recodes gender abbreviations in Big Class.jmp:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( sex =
   If(
      sex == "F", "Female",
      sex == "M", "Male",
      "Unknown" );
);
```

The script works like this:

`For Each Row( sex =`	For each row in the table, sex is the column that is recoded.
`If(`	Begins the If() loop.
`sex == "F", "Female",`	If the value of sex is F, replaces the value with Female.
`sex == "M", "Male",`	If the value of sex is M, replaces the value with Male.
`"Unknown" );`	If neither of the above conditions are true, replaces the value with Unknown. If this result were omitted and the value of sex were missing, the script would return a missing value.
`);`	Ends the loop.

You can also put actions and assignments in the result expression. The following example assigns 20 to x, because the first condition (y < 20) is false:

```
y = 25;
z = If( y < 20, x = y, x = 20 );
```

Note: Be careful to use two equal signs (==) for equality tests, not one equal sign (=). An `If`
with an argument such as *name=value* assigns rather than tests the value.

Avoiding Memory Issues with Multiple If Statements

Running a script that contains dozens of If() Statements can cause memory problems. We
recommend reducing the If() nesting. Suppose that your script contains 100 `If()` statements
in this format:

```
If( val, If( val, If( val, If( val, If( val,
If( val, If( val, If( val, If( val, If( val,
...
) ) ) ) ) ) ) ) ) );
```

Rewrite the script:

```
If( val, val1, If(val, val2, If( val, val3,
If( val, val4, If( val, val5, If( val, val6, ...) ) ) ) ) );
```

You can also try setting the maximum call depth. The `Maximum Call Depth` preference sets
the default for the maximum call depth (or stack size) in which JSL built-in functions,
user-defined functions, or `Recurse()` function calls can be made. By default, the maximum
call depth is set to 256.

In JMP, your JSL script has a limited amount of physical runtime stack memory in which to
perform JSL function calls. By default, this size is set to 2MB. Increasing the maximum call
depth can cause a physical runtime stack overflow, so incrementally increase this preference
in small amounts until you find the best value that works for your JSL script.

```
Preferences[1] << Set( Maximum JMP call depth( 64 ) );
```

Match

You can use the `Match()` function to make several equality comparisons without needing to
rewrite the value to be compared. The syntax is:

```
Match( x, value1, result1, value2, result2, ..., resultElse );
```

For example, the following script recodes gender abbreviations in Big Class.jmp:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( sex =
  Match(
    sex,
    "F", "Female",
    "M", "Male",
    "Unknown");
  );
```

The script works like this:

`For Each Row( sex =`	For each row in the table, sex is the column that is recoded.
`Match(`	Begins the `Match()` loop.
`sex,`	Specifies sex as the match argument.
`"F", "Female",`	If the value matches "F" replace it with "Female".
`"M", "Male",`	If the value matches "M", replace it with "Male".
`"Unknown" );`	If F or M are not matched, replaces the value with "Unknown".
`);`	Ends the loop.

This `Match()` example is a simplified version of the example in "If" on page 126. The advantage of `Match()` is that you define the comparison value once rather than repeat it in each condition. The disadvantage is that you cannot use expressions with operators as you can with `If`; the argument `sex == "F"` returns an error in a `Match()` expression.

With more groups of conditions and results, the value of `Match()` becomes more apparent. The following script would require many additional lines of code with `If()`.

```
dt = Open( "$SAMPLE_DATA/Travel Costs.jmp" );
For Each Row(
   Booking Day of Week = Match( Booking Day of Week,
      "Sunday", "SUN",
      "Monday", "MON",
      "Tuesday", "TUE",
      "Wednesday", "WED",
      "Thursday", "THU",
      "Friday", "FRI",
      "Saturday", "SAT",
      "Not Specified"
   )
);
```

Be careful with the data type of the condition and result. In the preceding example, the conditions and results are both character data. If the data types do not match, JMP automatically changes the column's data type. The results might not be what you want.

Note: The Match() function explicitly checks to see if the compare expression *x* is missing and if the value of *value1* is missing, then it returns the value of result1; otherwise it continues to compare the expression *x* to each *valueN* value in each *valueN/resultN* pair, ignoring any missing values. If the expression *x* is equal to any of the *valueN* value, then the corresponding *resultN* value is returned. If no matching *valueN* value is found, then the *resultElse* value is returned.

Choose

The Choose() function shortens scripts even more than Match(), provided the arguments are tested against integers. The syntax is:

```
Choose( expr, result1, result2, result3, ..., resultElse );
```

Suppose you have a data table with a group column of numeric values from 1 through 7. If the first cell contains the number 1, the following script returns x = "Low".

```
x = ( Choose( group[1], "Low", "Medium", "High", "Unknown" ); );
Show( x );
```

The script works like this:

x =	Creates the x variable.
Choose(	Begins the Choose() loop.
group[1],	Evaluates the value of group in the first row.
"Low",	If the value of group is 1, return "Low".
"Medium",	If the value of group is 2, return "Medium".
"High",	If the value of group is 3, return "High".
"Unknown"	Otherwise, return "Unknown".
)	Ends the loop.
);	Closes the x variable.
Show(x);	Returns the value of x.

If the expression evaluates to an out-of-range integer (such as 7 when only 4 replacement values are listed), the last result is returned. In the preceding example, "Unknown" is returned.

Notice that `If()` and `Match()` require more code to achieve the same results as the `Choose` function:

```
If(
    group[1] == 1, "Low",
    group[1] == 2, "Medium",
    group[1] == 3, "High",
    "Unknown"
);
Match( group, 1, "Low", 2, "Medium", 3, "High", "Unknown" );
```

Note: If the data types in the expression do not match, JMP automatically changes the column's data type.

Interpolate

The `Interpolate()` function performs linear interpolation for continuous data or bilinear interpolation for two-dimensional continuous data.

In the simplest case, the function finds the y value that corresponds to a given x value between two sets of points. You might use `Interpolate()` to calculate missing values between data points.

The data points can be specified as alternating ordered arguments:

```
Interpolate( x, x1, y1, x2, y2, ... );
```

or as matrices containing the x and y values:

```
Interpolate( x, xmatrix, ymatrix );
```

Suppose that your data set includes the height of individuals from age 20 through 25. However, there is no data point for age 23. To estimate the height for 23-year-olds, use interpolation. The following example shows the value that you want to evaluate (age 23), followed by matrices for ages (20 through 25) and heights (59 through 75).

```
Interpolate( 23, [20 21 22 24 25], [59 62 56 69 75] );
```

returns:

```
62.5
```

The value 62.5 is halfway between the y values 56 and 69, just as 23 is halfway between the x values 22 and 24.

You can also interpolate multiple points by specifying a matrix or list of values in the first argument.

```
Interpolate( [23, 24.5], [20 21 22 24 25], [59 62 56 69 75] );
```

returns:

```
[62.5, 72]
```

Bilinear Interpolation

You can also interpolate in two-dimensions:

```
Interpolate( {x, y}, xvector, yvector, zmatrix );
```

Here, the first argument is a list of two points, the second and third arguments are vectors that define the grid of x and y values, and the fourth argument is a matrix of data points. The function then finds the interpolated z value within the appropriate quadrant of the zmatrix. The appropriate quadrant is found by comparing the x and y values to the xvector and yvector arguments.

Suppose you have a 2x3 matrix and you want to interpolate a point that is halfway between the first and second rows of points and halfway between the second and third columns of points.

```
Interpolate( {0.5, 0.75}, [0 1], [0 0.5 1], [10 15 20, 12 16 18] );
```

returns:

```
17.25
```

Note that 0.5 is halfway between 0 and 1 for the x values (corresponding to the rows of the matrix) and 0.75 is halfway between 0.5 and 1 for the y values (corresponding to the columns of the matrix). You can find the bilinear interpolation in two steps:

1. Find the halfway points between the second and third columns for the each row of the matrix:
   – Row 1: 17.5 is halfway between 15 and 20.
   – Row 2: 17 is halfway between 16 and 18.
2. Find the halfway point between the two points in the previous step: 17.25.

Notes:

- The x values must be in ascending order. For example, `Interpolate(2,1,1,3,3)` returns 2. However, `Interpolate(2,3,3,1,1)` returns a missing value (.).
- `Interpolate()` is best used for continuous data, but `Step()` is designed for discrete data. See .

Step

The Step() is like Interpolate() except that it finds the corresponding y for a given x from a step-function fit rather than a linear fit. Use Step() with discrete y values (that is, when the y value's corresponding x value can be only y_1 or y_2). However, when the y value's corresponding x value can fall between y_1 and y_2, use Interpolate().

As with Interpolate, the data points can be specified as a list:

```
Step( x, x1, y1, x2, y2, ... );
```

or as matrices containing the x and y values:

```
Step( x, xmatrix, ymatrix );
```

Suppose that your data table shows the discount percentage for purchases of $25, $50, $75, and $100. You want to create a graph that shows the discount for a $35 purchase, which the data table does not specify. The following example shows the value that you want to evaluate, 35, followed by matrices for purchases from $25 to $100.

```
Step( 35, [25 50 75 100], [5 10 15 25] );
```

returns:

```
5
```

If the discounts were on a sliding scale (in this example, between 5 and 10, you would use Interpolate():

```
Interpolate( 35, [25 50 75 100], [5 10 15 25] );
```

returns:

```
7
```

As with Interpolate(), the data points must create a positive slope.

Compare Incomplete or Mismatched Data

Comparing data that contains missing values can return misleading results unless you specify a condition that is always true or use functions such as Is Missing() or Zero Or Missing(). Comparisons of data with mismatched types (numeric versus character) or data in matrices can also be confusing.

Table 5.5 shows examples of such comparisons and matrices and explanations of the results. For a review of operators used in comparisons, see "Operators" on page 103. For an overview of comparing data, see the *JSL Syntax Reference*.

Note: Matrices must include the same number of columns or rows.

Table 5.5 Some Special-Case Comparison Tests

Test	Result	Explanation
`m=.; m==1`	.	An equality test with a missing value returns missing.
`m=.; m!=1`	.	An inequality test with a missing value returns missing.
`m=.; m<1; m>1; and so on`	.	A comparison with a missing value returns missing (unless it could not possibly be true, see next).
`m=.; 1<m<0`	0	A comparison involving a missing value that could not possibly be true returns false; false takes precedence over missing for comparisons with more than two arguments (as with logical operators).
`{a, b}==List(a, b)`	1	An equality test of list arguments returns a single result.
`{a, b}<{a, c}`	.	A comparison test of list arguments is not allowed.
`1=="abc"`	0	An equality test with mixed data types returns false.
`1<="abc"`	.	A comparison with mixed data types returns missing.
`[1 2 3]==[2 2 5]`	[0 1 0]	An equality test of matrices returns a matrix of elementwise results. When a matrix is compared to a matrix, comparison is done element-by-element and returns a matrix of 1s and 0s.
`[1 2 3]==2`	[0 1 0]	An equality test of a matrix and a matrix filled with 2s. If a matrix is compared to a number, the number is treated as a matrix filled with that number.
`[1 2 3] < [2 2 5]`	[1 0 1]	A comparison of matrices returns a matrix of elementwise results.
`[1 2 3] < 2`	[1 0 0]	A comparison of a matrix and a matrix filled with 2s.
`Is Missing(m)`	1	Returns 1 for a missing value and returns 0 otherwise. For missing character values, you can also use empty quotes for the comparison, as in m == "".
`Zero Or Missing(m)`	1	Returns 1 when the value is 0 or missing. The argument must be numeric or a matrix and not a string.
`All([2 2]==[1 2])`	0	Summarizes elementwise comparisons; returns 1 only if *all* comparisons are true and returns 0 otherwise.

Table 5.5 Some Special-Case Comparison Tests *(Continued)*

Test	Result	Explanation
Any([2 2]==[1 2])	1	Summarizes elementwise comparisons; returns 1 if *any* comparison is true and returns 0 otherwise.

Missing Values

In a comparison, missing values typically return missing, not true or false. For this reason, it is very important to include a result that is always true. Suppose that a data table column contains the values 1, 2, 3, and a missing value in column A. A formula in column B sets up the comparison.

```
New Table( "Testing Comparisons",
    Add Rows( 4 ),
    New Column( "A",
        Numeric,
        "Continuous",
        Format( "Best", 10 ),
        Set Values( [1, 2, 3, .] )
    ),
    New Column( "B", Character, "Nominal", Formula( If( :A, "true", 1, "false"
    ) ) ) )
);
```

The following values are stored in column B:

```
"true"
"true"
"true"
"false"
```

The script works like this:

If(	Begins the comparison.
:A, "true",	If the value of A is nonmissing and nonzero, the result is "true". This comparison is true for the first three rows.
1, "false"	The value of 1 is always true, so the missing value returns "false".
);	Closes the comparison.

Notes:

- When a cell contains a non-empty string, the comparison fails with the first comparison. The following script returns a missing value if `a` is a string. It works for numeric values.

  ```
  a = "a";
  If( a, "true", "false" );
  ```

- When one value is false and another one is missing, `Or()` returns missing. Use `OrMZ()` to return false instead. See the *JSL Syntax Reference*.

- When one value is true and another one is missing, `And()` returns missing. Use `AndMZ()` to return false instead. See the *JSL Syntax Reference*.

- `MatchMZ()` is similar to `Match()` except that missing values are treated as 0.

- `IfMZ()` is similar to `If()` except that missing values are treated as 0.

Is Missing

If you know that some values are missing, you can also compare with `Is Missing()`. The comparison in the preceding example can be rewritten to return "missing" for missing values:

```
If( :A, "true", Is Missing( :A ), "missing", "false" );
```

The preceding expression returns "true" when A is nonmissing and nonzero, "missing" when A is missing, and "false" otherwise.

Zero Or Missing

If the missing value could be 0, use the `Zero Or Missing()` function instead:

```
Zero Or Missing( A );
```

This expression returns 1 when A is 0 or missing.

Tip: You cannot compare a known value with an explicitly defined missing value, only with variables, matrices, or other things that could *contain* missing values.

Inquiry Functions

Inquiry functions identify the type of an element, such as a string, list, or matrix. You can then write a script specific to that element type.

JMP also uses inquiry functions to determine the writability of a directory or file and to identify a computer's operating system and the JMP version.

General Element Types

The Type() function returns a string naming the type of the resulting value. For example:

```
Show( Type( 1 ), Type( "hi" ), Type( {"a", 2} ), Type( [10 24 325] ) );
```

results in:

```
Type(1) = "Integer"
Type("hi") = "String"
Type({"a", 2}) = "List"
Type([10 24 325]) = "Matrix";
```

Specific Element Types

Other inquiry functions (such as Is Matrix(), Is List(), Is Scriptable(), and so on) let you test for specific types of objects. In the following example, Is Matrix() evaluates as true, then the specified calculations are run:

```
a = [2 3];
b = [1, 1];
c = a * b;
If( Is Matrix( c ),
   (c ^ a) / (a * b),
   Print( "c is not a matrix." )
);
   [5 25]
```

Is Scriptable() returns 1 when the object is scriptable. Four variables in the following example refer to a data table, column, platform, and report. All four objects are scriptable, so Is Scriptable() returns 1 for each example.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( "weight" );
plat = Bivariate( Y( :weight ), X( :height ) );
rep = Report( plat );
Show( Is Scriptable( dt ) );
   Is Scriptable(dt) = 1;
Show( Is Scriptable( col ) );
   Is Scriptable(col) = 1;
Show( Is Scriptable( plat ) );
   Is Scriptable(plat) = 1;
Show( Is Scriptable( rep ) );
   Is Scriptable(rep) = 1;
```

Is Empty() tests to see whether a variable has a value, a function, an expression, or a reference to an object. It returns 1 if the variable is undefined or holds the Empty() value.Otherwise, you get errors when referring to something that has not been created or assigned a value yet. Programmers call this an *uninitialized variable*.

Here is an example of a test to see whether a data table is opened and therefore assigned to the dt variable. If a data table is not opened, the Open() function prompts the user to open the table.

```
If( Is Empty( dt = Current Data Table() ),
   dt = Open()
);
```

You can use Is Empty() for any variable (such as global variable, local variable, and columns).

Table 5.6 shows functions that identify object types.

Table 5.6 Inquiry Functions That Identify Object Types

Syntax	Explanation
Is Associative Array(*x*)	Returns 1 if the evaluated argument is an associative array or 0 otherwise.
Is Directory(x)	Returns 1 if the x argument is a directory and 0 otherwise.
Is Empty(*global*) Is Empty(*dt*) Is Empty(*col*)	Returns 1 if the global variable, data table, or data column does not have a value (is uninitialized) or 0 otherwise.
Is Expr(*x*)	Returns 1 if the evaluated argument is an expression or 0 otherwise.
Is File(x)	Returns 1 if the x argument is a file and 0 otherwise.
Is List(*x*)	Returns 1 if the evaluated argument is a list or 0 otherwise.
Is Matrix(*x*)	Returns 1 if the evaluated argument is a matrix or 0 otherwise.
Is Name(*x*)	Returns 1 if the evaluated argument is a name or 0 otherwise. See "Retrieve a Stored Expression Rather than its Result" on page 257.
Is Namespace(*x*)	Returns 1 if the evaluated argument is a namespace or 0 otherwise.
Is Number(*x*)	Returns 1 if the evaluated argument is a number or missing numeric value or 0 otherwise.
Is Scriptable(*x*)	Returns 1 if the evaluated argument is a scriptable object or 0 otherwise.
Is String(*x*)	Returns 1 if the evaluated argument is a string or 0 otherwise.
Type(*x*)	Returns a string naming the type of x.

Object Attributes

JMP provides the following functions to determine whether a file or directory is writable before attempting to write to them. Use these functions in combination with `Is Directory(path)` and `Is File(path)` to verify a script destination and attribute. See the *JSL Syntax Reference* for additional information.

The `Is Directory Writable(path)` function returns 1 if the directory specified in the path argument is writable and 0 otherwise.

The `Is File Writable(path)` function returns 1 if the file specified in the path argument is writable and 0 otherwise.

For example, the following code verifies the path refers to a directory and then checks to ensure the directory is writable:

```
If( Is Directory( "$SAMPLE_DATA/Loss Function Templates" ),
	If( Is Directory Writable( "$SAMPLE_DATA/Loss Function Templates" ),
		"Directory is writable.",
		"Directory is read only!"
	),
	"Is a read only directory."
);
```

Host Information

The `Host Is()` inquiry function identifies the current operating system. Then actions specific to that operating system can be performed.

The following example specifies text sizes in reports for different operating systems. If you commonly write your scripts on Windows and share them with macOS users, the results can look different from what you intended. For example, the following expression sets the text to a larger size (12) on macOS and a smaller size (10) on Windows:

```
textsize = If( Host is( "Mac" ), 12, 10 );
```

Version Information

The `JMP Version()` inquiry function returns the JMP version as a string. You might use this function to determine the JMP version and then run a script compatible with that version.

```
JMP Version(); // returns "16.0.0" in JMP 16
JMP Version(); // returns " 9.0.0" in JMP 9
```

Notice that a leading blank is inserted before versions less than 10.0.0. This blank helps when comparing version numbers. Without the leading blank, 9.0.0 is interpreted as greater than 10.0.0.

Chapter **6**

Types of Data

Working with Numbers, Strings, Dates, Currency, and More

This chapter discusses basic data types:

- numbers and strings
- paths, which are a special type of string
- dates and times, which can be either special numbers or special strings
- currency
- hexadecimal values and blobs

At the end of the chapter are two sections that show more advanced methods of interacting with strings and pattern matching with regular expressions.

Contents

Numbers and Strings

Numbers can be written as integers, decimal numbers, in scientific notation with an E preceding the power of ten, and as date-time values. A single period by itself is the missing numeric value.

For example, these are all numbers:

```
.   1   12   1.234  3E3  0.314159265E+1 1E-20
```

One or more characters placed within double quotation marks constitute a *string*. For example, these are all strings:

```
"Green" "Hello,\NWorld!" "54"
```

Notice that if a number is in quotation marks, it is a string, not a number. There are two functions you can use to change a number into a string or a string into a number.

- Use `Num()` to convert a string into a number. For example:

```
Num( "54" );
   54
```

> **Note:** `Num()` cannot convert non-numeric characters, so it produces a missing value.

```
Num( "Hello" );
   .
```

- Use `Char()` to convert a number into a string. For example:

```
Char( 54 );
   "54"
Char( 3E3 )
   "3000"
```

To preserve locale-specific numeric formatting in `Num()` or `Char()` output, include the `<<Use Locale(1)` option as shown in the following example:

```
Char( 42, 5, 2, << Use Locale( 1 ) );
   // results in the character value "42,00" in the French locale
```

To look at each character in a string, use the `Substr()` function. This example looks for the letter "a" in the string and prints a message to the log:

```
ch = Substr( "alphabetic", 1, 1 ); // start and end with the first character
If( ch == "a",
   Print( "First letter is a." )
);
   "First letter is a."
```

Formatting Numbers

The Format() function provides options for formatting numbers with thousands separators, decimal points, geographic symbols, *p*-values, and so on. The following example shows a basic format for a fixed decimal column.

```
Format( x, "Fixed Dec", 10, 2, "Use thousands separator");
```

The format starts with the number to be formatted. Then you can specify the width of the number, the number of decimal places, and whether to use a thousands separator.

Here are other examples:

```
Format( x, "Currency", 20, <<Use Locale(0) );
   // ignores computer locale and uses a period for the decimal separator
Format( x, "Currency", 20, "Use thousands separator" );
Format( x, "m/d/y", 10 );
Format( x, "Precision", 10, 2, "Keep trailing zeroes", "Keep all whole digits"
   );
Format( x, "Latitude DDD", "PUNDIR"); // "PUN" for punctuation, "DIR" for
   direction, PUNDIR for both
Format( x, "Custom", Formula( Abs( value ) ), 15 );
```

The syntax for Function() arguments is shown in the *JSL Syntax Reference*. You can also find the argument names and options in the data table Column Info window.

See "Date-Time Functions and Formats" on page 150 and "Currency" on page 164 for more information about formatting date-time values and currency.

Note: If the date format is unknown, an error is written to the log.

Matrix Box, Number Col Box, Number Col Edit Box, Number Edit Box

To format numbers in Matrix Box(), Number Col Box(), Number Col Edit Box(), and Number Edit Box(), send the Set Format message to the display box. You can then specify the same arguments as for the Format() function, such as the following:

```
<<Set Format( 10, 2, "Use thousands separator");
<<Set Format( "Currency", "EUR", 20 );
```

The syntax for Set Format arguments is shown in the *JSL Syntax Reference*. You can also find the argument names and options in the data table Column Info window.

Note: If you specify the number of decimal places, that number must be preceded with the width.

Unicode Characters

JMP supports both Unicode UTF-8 and UTF-16 standards for encoding and representing text for most of the world languages. See the The Unicode Consortium for code charts and more information on the Unicode standard.

To display Unicode characters in JMP, precede the Unicode code for the character with '\!'. For example:

- Greek letter sigma (σ) in Unicode = U+03C3; in JMP, use \!U03C3
- Greek letter mu (μ) in Unicode = U+03BC; in JMP, use \!U03BC

To use Unicode to express superscripts and subscripts:

- subscript 1 ($_1$) in Unicode = U+2081; in JMP, use \!U2081
- superscript 2 (^{2}) in Unicode = U+00B2; in JMP, use \!U00B2

To express x^2 in Unicode, in JMP, use \!U0078\!U00B2.

To use Unicode surrogate pairs, specify the Unicode escape sequence twice. If you specify an emoji Unicode character, you might need to change the font for it to appear correctly. The following example shows how to include an emoji in a text box.

```
New Window( "Emoji Example",
  V List Box(
    Text Box( "Surrogate Pair with space: \!UD83D \!UDE00", Set Font( "Segoe
  UI Emoji" ) ),
    Text Box( "Surrogate Pair without space: \!UD83D\!UDE00", Set Font(
  "Segoe UI Emoji" ) ) )
  )
);
```

Path Variables

Path variables are shortcuts to directories or files. Rather than enter the entire path to the directory or file, you use the path variable in a script. A path variable is a special type of string and is always contained within double quotation marks.

One common predefined path variable in JMP is $SAMPLE_DATA. This variable points to the sample data folder in your JMP or JMP Pro installation folder. The following example opens the Big Class.jmp sample data table.

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
```

Several path variables are predefined in JMP. The following table shows the definitions for the current JMP version. Variables in previous versions of JMP might differ.

Table 6.1 Predefined Path Variable Definitions

Variable	Path
ADDIN_HOME(com.your.addin.id) Defines the folder where the specified add-in is stored.	Add-In Builder stores the specified add-in in the following folders, based on your operating system: • Windows: "C:/Users/\<username\>/AppData/Roaming/SAS/JMP/Addins/" • macOS: "/Users/\<username\>/Library/Application Support/JMP/Addins/" For example, `Convert File Path( "$ADDIN_HOME(com.your.addin.id)` evaluates to the location of the `com.your.addin.id` add-in. To install your add-ins in a shared network location, use the `Register Addin()` function. See "Register an Add-In Using JSL" on page 895 in the "Creating Applications" chapter.
ALL_HOME Defines a folder that can be accessed by all users on a machine.	• Windows (JMP): "/C:/ProgramData/SAS/JMP/16/" • Windows (JMP Pro): "/C:/ProgramData/SAS/JMPPro/16/" • macOS: "/Library/Application Support/JMP/16/" To see if the directory exists, run the script `Is Directory("$ALL_HOME");`. If the folder exists, 1 is returned.
BUILTIN_SCRIPTS	• Windows (JMP): "/C:/Program Files/SAS/JMP/16/Resources/Builtins/" • Windows (JMP Pro): "/C:/Program Files/SAS/JMPPRO/16/Resources/Builtins/" • macOS: "/Applications/JMP 16.app/Contents/Resources/Builtins/"
DESKTOP	• Windows: "/C:/Users/\<username\>/Desktop/" • macOS "/Users/\<username\>/Desktop/"
DOCUMENTS	• Windows: "/C:/Users/\<username\>/Documents/" • macOS: "/Users/\<username\>/Documents/"
DOWNLOADS	• Windows: "/C:/Users/\<username\>/Downloads/" • macOS: "/Users/\<username\>/Downloads/"
GENOMICS_HOME	"/\<JMP Genomics installation directory\>/"

Table 6.1 Predefined Path Variable Definitions *(Continued)*

Variable	Path
HOME	• Windows (JMP): "/C:/Users/<username>/AppData/Roaming/SAS/JMP/16/" • Windows (JMP Pro): "/C:/Users/<username>/AppData/Roaming/SAS/JMPPro/16/" • macOS: "/Users/<username>/"
JMP_HOME	• Windows: /C:/<JMP installation directory> • macOS: "/Library/Application Support/JMP/16/JMP.app/Contents/Resources/AddIns/ **Note:** The macOS path exists only if add-ins are installed.
SAMPLE_APPS	• Windows: /C:/<JMP installation directory>/Samples/Apps/" • macOS: "/Library/Application Support/JMP/16/Samples/Apps/"
SAMPLE_DASHBOARDS	• Windows: /C:/<JMP installation directory>/Samples/Dashboards/" • macOS: "/Library/Application Support/JMP/16/Samples/Dashboards/"
SAMPLE_DATA	• Windows: "/C:/<JMP installation directory>/Samples/Data/" • macOS: "/Library/Application Support/JMP/16/Samples/Data/"
SAMPLE_IMAGES	• Windows: "/C:/<JMP installation directory>/Samples/Images/" • macOS "/Library/Application Support/JMP/16/Samples/Images/"
SAMPLE_IMPORT_DATA	• Windows: "/C:/<JMP installation directory>/Samples/Import Data/" • macOS: "/Library/Application Support/JMP/16/Samples/Import Data/"
SAMPLE_PROJECTS	• Windows: "/C:/<JMP installation directory>/Samples/Projects/" • macOS: "/Library/Application Support/JMP/16/Samples/Projects/"
SAMPLE_SCRIPTS	• Windows: "/C:/<JMP installation directory>/Samples/Scripts/" • macOS: "/Library/Application Support/JMP/16/Samples/Scripts/"
TEMP	• Windows: "/C:/Users/<username>/AppData/Local/Temp/" • macOS: "var/folders/..."

Table 6.1 Predefined Path Variable Definitions *(Continued)*

Variable	Path
USER_APPDATA Changes to JMP preferences, menus, and the Home Window are stored here, along with Debugger session settings.	• Windows (JMP): "/C:/Users/<username>/AppData/Roaming/SAS/JMP/16/" • Windows (JMP Pro): "/C:/Users/<username>/AppData/Roaming/SAS/JMPPro/16/" • macOS: "/Users/<username>/Library/Application Support/JMP/16/"

Path variable definitions are updated automatically based on the version of JMP you are using. For example, when you run a JMP 14 script in JMP 16, the JMP 16 path variable definitions are used.

To see the definition of any path variable, use the function `Get Path Variable`:

```
Get Path Variable( "HOME" );
    "/C:/Users/<username>/AppData/Roaming/SAS/JMP/16/"
```

Note that you don't include a dollar sign for `Set Path Variable()` or `Get Path Variable()`. But you must include the dollar sign when using the variable in a script.

Trailing Slashes

Make sure to include a trailing slash after the path variable. In the following example, the root name `"Big Class"` is assigned to the dtName variable. The `Open()` expression evaluates $SAMPLE_DATA and the trailing slash and then appends the dtName value along with the file extension `.jmp`.

```
dtName = "Big Class";
dt = Open( "$SAMPLE_DATA/" || dtName || ".jmp" );
```

The path is interpreted as:

```
C:/Program Files/SAS/JMP/16/Samples/Data/Big Class.jmp
```

Without the slash that follows $SAMPLE_DATA, the path is interpreted as:

```
C:/Program Files/SAS/JMP/16/Samples/DataBig Class.jmp
```

Create and Customize Path Variables

You can create your own path variables or override some of the built-in variables with the `Set Path Variable()`. In the following example, the path variable is called *root*. The variable points to the `c:/` directory.

```
Set Path Variable( "root", "c:/" );
```

To get the value of the new variable, use `Get Path Variable()`.

```
Get Path Variable( "root" );
    "c:/"
```

Use your path variable as you would other variables. The following expression opens the myimportdata.txt file in the c:/ directory.

```
Open( "$root/myimportdata.txt" );
```

As with getting path variables, omit the dollar sign when setting path variables.

Relative Paths

If you plan to use relative paths in variables, you must set the default directory. Then any path not preceded by a drive letter is relative to the default directory. Here is an example:

```
Set Default Directory( "c:/users/smith/data" );
```

To return the value of the default directory, use `Get Default Directory()`.

```
Get Default Directory(); // returns "c:/users/smith/data"
```

So the following expression resolves as C:/users/smith/data/cleansers.jmp:

```
Open("cleansers.jmp");
```

File Path Separators

In JMP, the preferred file path format is the Portable Operating System Interface (POSIX), or UNIX, format with forward slashes (/) as separators. This means that you do not have to identify the current operating system in scripts run on both Windows and macOS. However, each host still accepts its native format for compatibility.

You can a convert file path format from Windows to POSIX (and vice versa) using `Convert File Path()`. Converting from a POSIX to a Windows path might be useful when you need to output a path to a file or to another application. The syntax is:

```
Convert File Path (path, <absolute|relative>, <POSIX|windows>, <base(path)>);
```

For example, the following script converts a POSIX path to a Windows path:

```
Convert File Path( "c:/users/smith", windows);
    c:\users\smith
```

You can substitute a path variable (such as $HOME) for the path inside quotes.

Date-Time Functions and Formats

A date-time value consists of any portion of a date or time. The value can be seconds (3388594698), a complete date (such as "Wednesday, May 18, 2011"), the date and time ("05/18/2011 8:18:18 PM"), the week number (3), and so on.

JMP lets you convert date-time values to common formats, perform arithmetic on the values, and manipulate the data in a number of ways.

Tip: For descriptions of all date-time functions and their arguments, see the *JSL Syntax Reference*.

Date-Time Values

Date-time values are stored and calculated as the number of seconds since midnight, January 1, 1904. For example:

```
Today(); // returns 3388649872 on May 19, 2011 at 12:00:00 AM
```

As with `Today()`, the `Date DMY()` and `Date MDY()` functions also return month, day, and year arguments as seconds. For example, if it were 12:00:00 a.m. on May 19, 2011, all of the following statements would return the same value:

```
Date DMY( 19, 5, 2011 );
Date MDY( 5, 19, 2011 );
Today();
    3388608000
```

The `As Date()` function takes the number of seconds and displays it as a date or duration.

- Values that represent one year or more are returned as dates:

```
As Date( 3388608000 );
    19May2011
```

- Values that represent less than a year are returned as durations.

```
As Date( 50000 );
    :0:13:53:20
```

You can use date-time values in two ways:

- a literal value, for example `19May2011:10:10`
- a string, for example `"Thursday, May 19, 2011"`

You can perform arithmetic with date-time literals, which use the number of seconds as the base number.

```
As Date( 19May2011 + 1 );
```

19May2011:00:00:01

Program with Date-Time Functions

Table 6.2 shows functions that convert seconds into date-time values and date-time values into seconds.

Table 6.2 Date-Time Functions

Function	Explanation
Abbrev Date(*date*)	Returns a string representation for the *date* supplied. The format is based on your computer's regional setting. So for the English (United States) locale, the date is formatted like **"02/29/2004"**. Even if you are running JMP in English with a different locale, the locale format is applied.
As Date(*expression*)	Formats a number or expression so that it shows as a date or duration in a text window. For example, values that represent one year or more are returned as dates. x = As Date(8Dec2000 + In Days(2)); shows as: *10Dec2000* Values that represent less than a year are returned as durations. As Date(50000); shows as: *:0:13:53:20*
Date DMY(*day*, *month*, *year*)	Returns the specified date expressed as the number of seconds since midnight, 1 January 1904. For example, the second Leap Day of the third millennium is DateDMY(29,2,2004), which returns 3160857600.
Date MDY(*month*, *day*, *year*)	Returns the specified date expressed as the number of seconds since midnight, 1 January 1904. For example, the second Leap Day of the third millennium is DateMDY(2,29,2004), which returns 3160857600.
Day Of Week(*date*)	Returns an integer representation for the day of the week of the *date* supplied. Weeks are Sunday–Saturday.

Table 6.2 Date-Time Functions *(Continued)*

Function	Explanation
Day Of Year(*date*)	Returns an integer representation for the day of the year of the *date* supplied.
Day(*date*)	Returns an integer representation for the day of the month of the *date* supplied.
Format(*date*, "*format*")	Returns the *value* in the *format* specified in the second argument. Most typically used for formatting datetime values from a number of seconds to a formatted date. Format choices are those shown in the **Column Info** dialog box. Also see Table 6.3 "How JMP Interprets Two-Digit Years" on page 157.
Hour(*datetime*)	Returns an integer representation for the hour part of the *date-time* value supplied.
In Days(*n*) In Hours(*n*) In Minutes(*n*) In Weeks(*n*) In Years(*n*)	These functions return the number of seconds per *n* minutes, hours, days, weeks, or years. Divide by these functions to express an interval in seconds as an interval in other units.
Long Date(*date*)	Returns a string representation for the specified *date*. The format is based on your computer's regional setting. So for the English (United States) locale, the date is formatted like "Sunday, February 29, 2004". Even if you are running JMP in English with a different locale, the locale format is applied.
MDYHMS(*date*)	Returns a string representation for the *date* supplied, formatted like "2/29/2004 00:02:20 AM".
Minute(*date-time*)	Returns an integer representation for the minute part of the *date-time* value supplied.
Month(*date*)	Returns an integer representation for the month of the *date* supplied.
Num(*date-time*)	Returns an integer representation for the *date-time* value supplied.
InFormat(*string*, "*format*") Parse Date(*string*, "*format*")	Parses a *string* of a given *format* and returns datetime value expressed as if surrounded by As Date(), returning the date in ddMonyyyy format.

Table 6.2 Date-Time Functions *(Continued)*

Function	Explanation
Second(*date-time*)	Returns an integer representation for the second part of the *date-time* value supplied.
Short Date(*date*)	Returns a string representation for the *date* supplied, in the format MM/DD/YYYY, regardless of locale (for example, "02/29/2004").
Time Of Day(*date*)	Returns an integer representation for the time of day of the *date-time* supplied.
Today()	Returns the current date and time expressed as the number of seconds since midnight, 1 January 1904. No arguments are accepted, but the parentheses are still needed.
Week Of Year(*date*, *<rule_n>*)	Returns the week of the year as a date-time value. Three rules determine when the first week of the year begins. • With rule 1 (the default), weeks start on Sunday, with the first Sunday of the year being week 2. Week 1 is a partial week or empty. • With rule 2, the first Sunday begins with week 1, with previous days being week 0. • With rule 3, the ISO-8601 week number is returned. Weeks start on Monday. Week 1 is the first week of the year with four days in that year. It is possible for the first or last three days of the year to belong to the neighboring year's week number.
Year(*date*)	Returns an integer representation for the year of the specified *date*.

Examples of Common Date-Time Functions

You can use any function that returns seconds within a function that returns a date-time.

For example, if today is May 19, 2011 and the time is 11:37:52 AM, Today() returns the number of seconds, and the functions that follow show that number of seconds since the base time in different date-time formats:

```
Today()
    3666955847
Short Date( Today() );
    "03/13/2020"
```

```
Long Date( Today() );
   "Friday, March 13, 2020"
Abbrev Date( Today() );
   "3/13/2020"
MDYHMS( Today() );
   "03/13/2020 2:50:17 PM"
```

The date argument in parentheses can be seconds (or any function that returns seconds), or any date-time literal value. For example, both of the following expressions return the same value:

```
Long Date( 3388649872 );
Long Date( 19May2011 );
   "Friday, March 13, 2020"
```

Note: Long Date() and Abbrev Date() values are formatted according to your computer's regional settings.

Extract Parts of Dates

You can extract parts of date values using the functions Month(), Day(), Year(), Day Of Week(), Day Of Year(), Week Of Year(), Time Of Day, Hour(), Minute(), and Second(), which all return integers. If today is May 24th, 2011, each of the following examples returns the 144th day of the year:

```
Day of Year( Today() );
Day of Year( 24May2011 );
Day of Year( Date MDY( 5, 24, 2011 ) );
   144
```

Example

A data table column named Date contains date-time values that are formatted as "m/d/y". You want to create a column that shows only the time. In the following script, the second column's formula extracts the time of day from the Date value in the first column.

```
New Table( "Assembly Tests",
   Add Rows( 1 ),
   New Column( "Date",
      Numeric, Continuous,
      Format( "m/d/y" ),
      Set Values( [3389083557] )
   ),
   New Column( "Time",
      Numeric, Continuous,
      Formula( Format( Time Of Day( :Date ), "h:m:s") )
   )
);
```

Figure 6.1 shows the result. Note that the time of day does not appear in the Date column, because the Format function applies the "m/d/y" format.

Figure 6.1 Example of Extracting the Time

Rules for Determining the Week of the Year

Week of Year() returns the week of the year as a date-time value. Three rules determine when the first week of the year begins.

- With rule 1 (the default), weeks start on Sunday, with the first Sunday of the year being week 2. Week 1 is a partial week or empty.

  ```
  Week Of Year( Date DMY( 19, 6, 2013 ), 1 );
      25
  ```

- With rule 2, the first Sunday begins with week 1, with previous days being week 0.

  ```
  Week Of Year( Date DMY( 19, 6, 2013), 2 ) ;
      24
  ```

- With rule 3, the ISO-8601 week number is returned. Weeks start on Monday. Week 1 is the first week of the year with four days in that year. It is possible for the first or last three days of the year to belong to the neighboring year's week number.

  ```
  Week Of Year( Date DMY( 19, 6, 2013 ), 3 );
      25
  ```

Arithmetic on Dates

You can perform the usual arithmetic operations with date-time data as with any other numeric data. One option is simple arithmetic, such as subtracting a number from a date-time value.

Another option is writing a formula to perform the arithmetic.

Example

The Date column in your data table shows when a customer uses his credit card to buy gas. You want to know how many days elapse between purchases. The following script creates a Days elapsed column. The formula in that column subtracts the Date value in the current row from that of the previous row.

```
New Table( "Gas Purchases",
    Add Rows( 3 ),
    New Column( "Date",
```

```
        Numeric,
        "Continuous",
        Format( "m/d/y" ),
        Set Values( [3392323200 3393532800 3394828800] )
    ),
    New Column( "Days elapsed",
        Formula(
        // returns a missing value for the first row
            If( Row() == 1,
                .,
                (:Date[Row()] - :Date[Row() - 1]) / In Days()
            )
        )
    )
);
```

Figure 6.2 shows the result.

Figure 6.2 Example of Calculating Date-Time Values

	Date	Days elapsed
1	07/01/2011	•
2	07/15/2011	14
3	07/30/2011	15

Time Intervals

The In Minutes, In Hours, In Days, In Weeks, and In Years functions are used to express time intervals in units rather than seconds. Each of these functions returns the number of seconds associated with a particular period of time. For example, the following expression returns the number of weeks between now and July 4, 2012.

```
(Date DMY( 04, 07, 2012 ) - Today()) / In Weeks();
    -208.225444775132
```

When the argument for the interval function is empty, JMP counts by 1. You can enter another number to change the count. For example, In Years(10) converts the interval to decades. The following expression returns the number of decades between now and December 31, 2037.

```
(Date DMY( 31, 12, 2037) - Today() ) / In Years( 10 );
    2.18927583529799
```

Two-Digit and Four-Digit Years

JMP applies its own algorithms for interpreting and displaying datetime strings rather than supporting operating system-specific datetime formats. However, JMP uses the date-time separators selected in the Region and Language control panel (Windows) or the Date & Time preferences (macOS) to interpret and display dates.

Two-digit years are interpreted according to the current system clock year and JMP rules. For example, when the year in a script is 11, and you run the script after 1990, the year shows as 2011.

```
Long Date( 25May11 );
    "Wednesday, May 25, 2011"
```

To avoid ambiguity, enter four-digit years. The following expression returns 1911 (rather than 2011) as indicated:

```
Long Date( 25May1911 );
    "Thursday, May 25, 1911"
```

Table 6.3 explains how JMP interprets two-digit years.

Table 6.3 How JMP Interprets Two-Digit Years

Two-Digit Year Value	When it is Evaluated	Result	Examples	Result
00–10	before 1990 (on Windows)	19__	enter 5 in year 1979	1905
	before or during 1990 (on macOS)			
	during or after 1990 (on Windows)	20__	enter 5 in year 1991	2005
	after 1990 (on macOS)			
11–89 (on Windows)	any time	current century	enter 13 in year 1988	1913
11–90 (on macOS)			enter 13 in year 2024	2013
90–99 (on Windows)	before 2011	19__	enter 99 in year 1999	1999
91–99 (on macOS)	during or after 2011	20__	enter 99 in year 2015	2099

Note: JMP always displays four-digit years regardless of the regional settings. If you need to show two-digit years, use character string functions. See the "Types of Data" chapter on page 141.

Date-Time Values in Data Tables

Change Date-Time Input and Display Formats

In data tables, JMP can accept the input of date-time values in one format (the *input format*), store them internally as the number of seconds since the base date, and display them in a different date-time format. The Informat() and Format() functions give you this control.

- Informat() takes a string date-time value, defines the date format used in that string, and returns the date in ddMonyyyy format.

  ```
  Informat( "19May2011 11:37:52 AM", "ddMonyyyy h:m:s" );
      19May2011:11:37:52
  ```

- Format() takes the number of seconds since the base date (or a date-time function that returns that number) and returns the date in the specified format.

  ```
  Format( 3388649872, "ddMonyyyy h:m:s" );
      "19May2011 11:37:52 AM"
  Format( Today(), "ddMonyyyy h:m:s" );
      "19May2011 11:37:52 AM"
  ```

Suppose that you are entering dates into a column using the d/m/y h:m format, but you want to see the dates in the m/d/y format. Input Format defines the input format, and Format defines the display format. For example,

```
New Table( "Widget Assembly",
    Add Rows( 1 ),
    New Column( "Date",
        Numeric,
        "Continuous",
        Format( "m/d/y" ),
        Input Format( "d/m/y h:m" ),
        Set Values( [3126917100] )
    )
);
```

The Format and Input Format values are shown in the data table's column properties (Figure 6.3). Note that when you click in the cell to edit it, the date-time value appears in the input format. When you edit the value, or add a new value, the format specified in the data table column Format list is used to display the value.

Figure 6.3 Example of Date-Time Display and Input Values
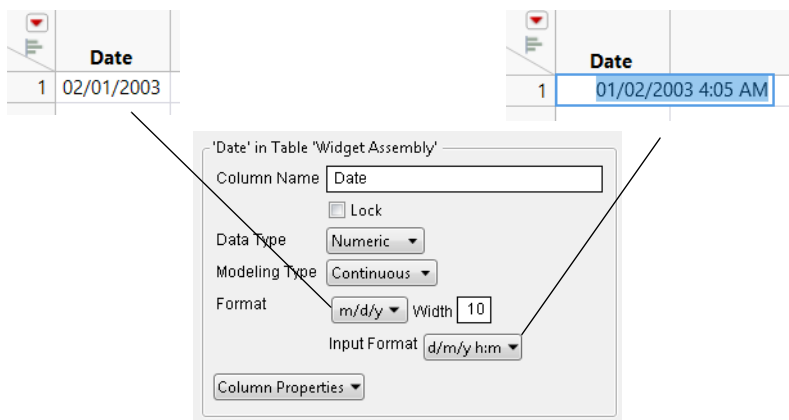
Notes:

- In a script that converts a column from character to numeric, specify Format() and Informat() to prevent missing values. See "Convert Character Dates to Numeric Dates" on page 902 in the "Common Tasks" chapter.

- The date-separator character on your computer might differ from the forward slash (/) character shown in the *Scripting Guide*.

- You can enter time values in 24-hour format (military time) or with AM or PM designators.

Table 6.4 describes the formats used as arguments in date-time functions or as data table formats. You can also use the formats for the *format* argument to a Format message to a data column. See "Set or Get Formats" on page 416 in the "Data Tables" chapter.

For descriptions of specific date-time functions, see the *JSL Syntax Reference*.

Notes:

- Date-time formats in which a colon or T separate the date and time always use a 24-hour format. Other formats use your computer's regional settings.

- If the format is unknown, an error is written to the log.

Table 6.4 Date-Time Formats

Type	Format argument	Example
Date only	"m/d/y"	"01/02/1999"
	"mmddyyyy"	"01021999"
	"m/y"	"01/1999"

Table 6.4 Date-Time Formats *(Continued)*

Type	Format argument	Example
	"d/m/y"	"02/01/1999"
	"ddmmyyyy"	"02011999"
	"ddMonyyyy"	"02Jan1999"
	"Monddyyyy"	"Jan021999"
	"y/m/d"	"1999/01/02"
	"yyyymmdd"	"19990102"
	"yyyy-mm-dd"	"1999-01-02"
	"yyyyQq"	1999Q1
Date and time	"m/d/y h:m"	"01/02/1999 13:01" "01/02/1999 1:01 PM"
	"m/d/y h:m:s"	"01/02/1999 13:01:55" "01/02/1999 1:01:55 PM"
	"d/m/y h:m"	"02/01/1999 13:01" "02/01/1999 1:01 PM"
	"d/m/y h:m:s"	"02/01/1999 13:01:55" "02/01/1999 1:01:55 PM"
	"y/m/d h:m"	'1999/01/02 13:01' '1999/01/02 1:01 PM'
	"y/m/d h:m:s"	'1999/01/02 13:01:02 '1999/01/02 1:01:02 PM'
	"ddMonyyyy h:m"	"02Jan1999 13:01" "02Jan1999 1:01 PM"
	"ddMonyyyy h:m:s"	"02Jan1999 13:01:02" "02Jan1999 1:01:02 PM"
	"ddMonyyyy:h:m"	"02Jan1999:13:01" "02Jan1999:1:01 PM"
	"ddMonyyyy:h:m:s"	"02Jan1999:13:01:02" "02Jan1999:1:01:02 PM"
	"Monddyyyy h:m"	"Jan021999 13:01" "Jan021999 1:01 PM"
	"Monddyyyy h:m:s"	"Jan021999 13:01:02" "Jan021999 1:01:02 PM"

Table 6.4 Date-Time Formats *(Continued)*

Type	Format argument	Example
Day number and time	`":day:hr:m"`	`"34700:13:01"` `":33:001:01 PM"`
	`":day:hr:m:s"`	`"34700:13:01:02"` `":33:001:01:02 PM"`
	`"h:m:s"`	`"13:01:02"` `"01:01:02 PM"`
	`"h:m"`	`"13:01"` `"01:02 PM"`
	`"yyyy-mm-ddThh:mm"`	`1999-01-02T13:01`
	`"yyyy-mm-ddThh:mm:ss"`	`1999-01-02T13:01:02`
Duration	`":day:hr:m"`	"52:03:01" reads fifty-two days, three hours, and one minute
	`":day:hr:m:s"`	"52:03:01:30" reads fifty-two days, three hours, one minute, and thirty seconds
	`"hr:m"`	"17:37" reads seventeen hours and thirty-seven minutes
	`"hr:m:s"`	"17:37:04" reads seventeen hours, thirty-seven minutes, and 4 seconds
	`"min:s"`	"37:04" reads thirty-seven minutes and 4 seconds

Note: The following formats display the date-time according to your computer's regional settings. They are available only for the display of dates, not for date input in a data table. Examples are shown for the United States locale.

Type	Format argument	Example
Abbreviated date	`"Date Abbrev"`	(Display only) "01/02/1999"
Long date	`"Date Long"`	(Display only) "Saturday, January 02, 1999"
Locale date	"Locale Date"	(Display only) "01/02/1999"

Table 6.4 Date-Time Formats *(Continued)*

Type	Format argument	Example
Locale date and time	"Locale Date Time h:m"	(Display only) "01/02/1999 13:01" or "01/02/1999 01:01 PM"
	"Locale Date Time h:m:s"	(Display only) "01/02/1999 13:01:02" or "01/02/1999 01:01:02 PM"

Custom Date-Time Formatting with Format Patterns

Format patterns are strings that define a date-time format. Here's one example:

`<YYYY></><MM></><DD> <hh><:><mm><:><ss><ampm>`

The parts of the pattern in angle brackets are called *field descriptors*. They represent a value (such as <YYYY>, which is a four-digit year) or other date-time text (such as </>, which is a locale-specific date separator).

The conversion of a date-time value to a string and a string converted to a date-time value work as follows:

- When a date-time value is converted to a string, the field descriptors are replaced by the appropriate value or symbols.

- When a string is converted to a date-time value, a value is or matching symbols are expected to be found where each field descriptor is. Anything outside a format descriptor is treated as literal text. The literal text is replaced as-is when it is converted *to* a string, and is expected to be found where specified when converting *from* a string.

Table 6.5 Output for Custom Date-Time Formats

Format	Output
`<YYYY></><MM></><DD> <hh><:><mm><:><ss><ampm>`	2020/02/10 11:54:33 AM
`<DayOfWeek>, <Month> <D>, <YYYY>`	Wednesday, February 5, 2020
`<Day>d <hh24>h <mm>m <ss>s`	123d 02h 34m 56s

Note: <Day> is the day count, used as the most significant field of a duration. <Hour> and <Minute> field descriptors are also available.

Example of Outputting the Date-Time Value to the Log

The following script uses `Informat()` to read a string date-time value, defines the format pattern used in that string, and returns the date in **ddMonyyyy** format. The script then outputs the date-time value to the log.

```
dateVal = Informat( "June 16, 2016", "Format Pattern", "<Month> <D>, <YYYY>"
   );
   16Jun2016
```

Here's another way to specify the format pattern:

```
date2 = 16Jun2021;

dayofwk2 = Format( date2, "Format Pattern", "<DayOfWeek>" );
monthnew2 = Format( date2, "Format Pattern", "<Month>" );

YYYYMMDD = Format( date2, "Format Pattern", "<YYYY></><MM></><DD>" );
YYMMMDD = Format( date2, "Format Pattern", "<YY></><MMM></><DD>" );
   "21/Jun/16"
```

Example of Changing the Input Format of a Column

The following script shows how to change the input format of a column. The Input Format appears when you edit the value in a data table.

```
New Table( "temp_date",
   Add Rows( 2 ),
   New Column( "date1",
      Numeric,
      "Continuous",
      // specify how the value appears in the data table
      Format( "Format Pattern", "<Month> <D>, <YYYY>", 35 ),
      // specify how the value appears when you edit the value
      Input Format( "Format Pattern", "<YYYY>*<MM>*<DD>" ),
      Set Values( {"2018*10*31", "2018*09*09"} )
   ),
   New Column( "date2",
      Numeric,
      "Continuous",
      Format( "Format Pattern", "<YYYY>*<MM>*<DD>", 10 ),
      Set Values( {"2018*02*05", "2018*07*03"} )
   )
);
```

Currency

JMP displays numbers as currency using the `Format()` function, which uses the following syntax:

```
Format(x,"Currency", <"currency code">, <decimal>, < << Use Locale(Boolean));
```

Where:

- *x* is a column or a number
- *"currency code"* is an International Standards Organization (ISO) 4217 code
- *decimal* is the number of decimal places

To illustrate the `Format` function:

```
Format( 12345.6, "Currency", "GBP", 3 );
   "£12,345.600"
```

By default, the operating system locale is used for the currency if the currency code symbol is omitted. For example, running the following script in a Japanese operating system formats the number with the yen symbol.

```
Format( 12345.6, "Currency", 3);
   "¥12,345.600"
```

To ignore the computer's locale and use a period for the decimal separator, include the `Use Locale(0)` argument:

```
Format( 12345.6, "Currency", 3, <<Use Locale( 0 ) );
```

Tip: If you change the region when JMP is open, restart JMP for the `Use Locale` setting to work properly.

If the currency code is not supported by JMP, the currency code string appears before the number.

```
Format( 12345.6, "Currency", "BBD", 3 );
   "BBD 12,345.600"
```

Table 6.6 lists the currencies supported in JMP.

Table 6.6 Currencies Supported in JMP

Code	Currency	Code	Currency	Code	Currency
AUD	Australian dollar	ILS	Israeli new shekel	RUB	Russian ruble
BRL	Brazilian real	INR	Indian rupee	SEK	Swedish krone

Table 6.6 Currencies Supported in JMP *(Continued)*

Code	Currency	Code	Currency	Code	Currency
CAD	Canadian dollar	JPY	Japanese yen	SGD	Singapore dollar
CHF	Swiss franc	KRW	South Korean won	THB	Thai baht
CNY	Chinese yuan	MXN	Mexican peso	TRY	New Turkish lira
COP	Colombian peso	MYR	Malaysian ringgit	TWD	New Taiwan dollar
DKK	Danish krone	NOK	Norwegian krone	USD	US dollar
EUR	Euro	NZD	New Zealand dollar	ZAR	South African rand
GBP	British pound	PHP	Philippine peso		
HKD	Hong Kong dollar	PLN	Polish zloty		

For more information about how to format currency, see "Formatting Numbers" on page 144.

Hexadecimal and BLOB Functions

JMP can also handle binary (large) objects, commonly called BLOBs. The functions below convert between hexadecimal values, numbers, characters, and BLOBs. Some of the functions are covered in more detail following Table 6.7.

See the *JSL Syntax Reference* for more information.

Table 6.7 Hexadecimal and BLOB Functions

Syntax	Explanation
Hex("*text*") Hex("*num*") Hex("*blob*")	Returns the hexadecimal codes for the characters in *text*, *number*, or *blob*. Char To Hex is an alias.
Hex To BLOB("*hexstring*")	Returns a BLOB representation of the hexadecimal code supplied as a quoted string.

Table 6.7 Hexadecimal and BLOB Functions *(Continued)*

Syntax	Explanation
Hex To Char("*hexstring*", "*encoding*")	Returns a character string that corresponds to the hexadecimal code supplied as a quoted string.
	The default encoding for the character string is utf-8. utf-16le, utf-16be, us-ascii, iso-8859-1, ascii~hex, shift-jis, and euc-jp are also supported.
Hex to Number	Returns the number that corresponds to the hexadecimal code suppled as a quoted string.
Char To BLOB("*string*") Char To BLOB("*string*", "*encoding*")	Converts a character string into a binary (BLOB).
	The default encoding for the blob is utf-8. utf-16le, utf-16be, us-ascii, iso-8859-1, ascii~hex, shift-jis, and euc-jp are also supported.
BLOB To Char("*blob*") BLOB To Char("*blob*", "*encoding*")	Converts binary data to a character string.
	The default encoding for the character string is utf-8. utf-16le, utf-16be, us-ascii, iso-8859-1, ascii~hex, shift-jis, and euc-jp are also supported.
BLOB Peek("*blob*", *offset*, *length*)	Returns a new BLOB that is a subset of the given BLOB that is *length* bytes long and begins at the *offset*. Note that the offset is 0-based.
Matrix to BLOB(matrix, type, bytesEach, endian)	Makes a BLOB from a matrix by converting the matrix elements to 1-byte, 2-byte, or 4-byte signed or unsigned integers or 4-byte or 8-byte floating point numbers.
BLOB to Matrix(blob, "type", bytes, "endian", <nCols>)	Creates a matrix by converting each byte in the blob to numbers.

Hex(string) returns the hexadecimal codes for each character in the argument. For example,

```
Hex( "Abc" );
```

returns

```
"416263"
```

since 41, 62, and 63 are the hexadecimal codes (in ASCII) for "A", "b", and "c".

Hex to Char(string) converts hexadecimal to characters. The resulting character string might not be valid display characters. All the characters must be in pairs, in the ranges 0-9, A-Z, and a-z. Blanks and commas are allowed, and skipped. For example,

```
Hex To Char( "4142" );
```

returns

```
"AB"
```

since 41 and 42 are the hexadecimal equivalents of "A" and "B".

Hex and `Hex To Char` are inverses of each other, so

```
Hex To Char( Hex( "Abc" ) );
```

returns

```
"Abc"
```

`Hex To BLOB(string)` takes a string of hexadecimal codes and converts it to a binary object.

```
a = Hex To BLOB( "6A6B6C" );
Show( a );
  a = Char To BLOB("jkl", "ascii~hex")
```

`BLOB Peek(blob,offset,length)` extracts bytes as defined by the arguments from a blob.

```
b = BLOB Peek( a, 1, 2 );
Show( b );
  b = Char To BLOB("kl", "ascii~hex")
b = BLOB Peek( a, 0, 2 );
Show( b );
  b = Char To BLOB("jk", "ascii~hex")
b = BLOB Peek( a, 2 );
Show( b );
  b = Char To BLOB("l", "ascii~hex")
```

`Hex(blob)` converts a blob into hexadecimal.

```
c = Hex( a );
Show( c );
  c = "6A6B6C"
d = Hex To Char( c );
Show( d );
  d = "jkl"
```

`Concat(blob1,blob2)` or `blob1 || blob2` concatenates two blobs.

```
e = Hex To BLOB( "6D6E6F" );
Show( e );
f = a||e;
Show( f );
  e = Char To BLOB("mno", "ascii~hex")
  f = Char To BLOB("jklmno", "ascii~hex")
```

`Length(blob)` returns the number of bytes in a blob.

```
g = Length( f );
```

```
Show( g );
   g = 6
```

Note: When blobs are listed in the log, they are shown with the constructor function
`Char To BLOB("...")`.

Any hex code outside the ASCII range (`space` to }, or hex 20 - 7D) is encoded as the
three-character sequence [~][hexdigit][hexdigit]. For example,

```
h = Hex To BLOB( "19207D7E" );
Show( h );
i = Hex( h );
Show( i );
   h = Char To BLOB("~19 }~7E", "ascii~hex")
   i = "19207D7E"
```

`Char To BLOB(string)` creates a blob from a string, converting ~hex codes.
`BLOB To Char(blob)` creates a string with ~hex codes to indicate non-visible and non-ASCII
codes.

Work with Character Functions

This section shows how to use some of the more complex character functions that are
described in the *JSL Syntax Reference*.

Concat

In the `Concat` function, expressions yielding names are treated like character strings, but
globals that have the name values are evaluated. The following example demonstrates that if
you have a stored name value, you need to either use `Char` before storing it in a global, or `Name
Expr` on the global name.

```
n = {abc};
c = n[1] || "def";
Show( c );
   "abcdef"

m = Expr( mno );
c = m || "xyz";
Show( c );
   Name Unresolved: mno in access or evaluation of 'mno' , mno/*###*/
// The /*###*/ characters indicate the location of an error in the script
   execution.

m = Expr( mno );
```

```
c = Name Expr( m ) || "xyz";
Show( c );
   "mnoxyz"

m = Char( Expr( mno ) );
c = m || "xyz";
Show( c );
   "mnoxyz"
```

Concat Items() converts a list of string expressions into a single string, with each item separated by a delimiter. If unspecified, the delimiter is a blank. Its syntax is

```
resultString = Concat Items ({list of strings}, <"delimiter string">);
```

For example,

```
a = {"ABC", "DEF", "HIJ"};
result = Concat Items(a, "/");
```

returns

```
"ABC/DEF/HIJ"
```

Alternatively,

```
result = Concat Items( a );
```

returns

```
"ABC DEF HIJ"
```

Munger

Munger works many different ways, depending on what you specify for its arguments:

```
Munger(string, offset, find | length, <replace>);
```

Table 6.8 Munger behaviors for various types of arguments

Find, *length*, and *replace* arguments	**Example**
If you specify a string as the *find* and specify no *replace* string, Munger returns the position (after *offset*) of the first occurrence *find* string.	Munger("the quick brown fox", 1, "quick"); *5*
If you specify a positive integer as the *length* and specify no *replace* string, Munger returns the characters from *offset* to *offset* + *length*.	Munger("the quick brown fox", 1, 5); *"the q"*

Table 6.8 Munger behaviors for various types of arguments *(Continued)*

Find, *length*, and *replace* arguments	Example
If you specify a string as the *find* and specify a *replace* string, Munger replaces the first occurrence after *offset* of *text* with *replace*.	```Munger("the quick brown fox", 1, "quick", "fast");``` *"the fast brown fox"*
If you specify a positive integer as the *length* and specify a *replace* string, Munger replaces the characters from *offset* to *offset* + *length* with *replace*.	```Munger("the quick brown fox", 1, 5, "fast");``` *"fastuick brown fox"*

If you specify a positive integer as the *length*, and *offset* + *length* exceeds the length of *text*, Munger either returns *text* from *offset* to the end or replaces that portion of *text* with the *replace* string, if it exists.	```Munger("the quick brown fox", 5, 25);```
	 "quick brown fox"
	 ```Munger("the quick brown fox", 5, 25, "fast");```
	 "the fast"

If you specify zero as the *length* and specify no *replace* string, Munger returns a blank string.	```Munger("the quick brown fox", 1, 0);``` *""*
If you specify zero as the *length* and specify a *replace* string, the string is inserted before the *offset* position.	```Munger("the quick brown fox", 1, 0, "see");``` *"see the quick brown fox"*
If you specify a negative integer as the *length* value and specify no *replace* string, Munger returns all characters from the offset to the end of the string.	```Munger("the quick brown fox", 5, -5);``` *"quick brown fox"*
If you specify a negative integer for *length* and specify a *replace* string, Munger replaces all characters from the offset to the end with the *replace* string.	```Munger("the quick brown fox", 5, -5, "fast");``` *"the fast"*

Repeat

The Repeat() function makes copies of its first argument into a result. The second (and sometimes a third) argument is the number of repeats, where 1 means a single copy.

If the first argument evaluates to a character value or list, the result is that many copies.

```
Repeat( "abc", 2 );
```

```
    "abcabc"
Repeat( {"A"}, 2 );
    {"A","A"}
Repeat( {1, 2, 3}, 2 );
    {1,2,3,1,2,3}
```

If the first argument evaluates to a number or matrix, the result is a matrix. The second argument is the number of row repeats, and a third argument can specify the number of column repeats. If only two arguments are specified, the number of column repeats is 1.

```
Repeat( [1 2, 3 4], 2, 3 );
    [ 1 2 1 2 1 2,
      3 4 3 4 3 4,
      1 2 1 2 1 2,
      3 4 3 4 3 4]
Repeat( 9, 2, 3 );
    [ 9 9 9,
      9 9 9]
```

The repeat function is compatible with the function of the same name in the SAS/IML language, but is incompatible with the SAS character DATA step function, which repeats one more time than this function.

Substitute and Substitute Into

The Substitute() function returns a copy of a string with a replacement expression. The Substitute Into() function changes the original string with a replacement expression.

For example, consider the following script:

```
str1 = str2 = "All things considered";

// str3 holds the result of the Substitute, and str1 is not changed
str3 = Substitute( str1, "All", "Some" );

/* Substitute Into returns nothing, so str4 is missing,
and str2 is changed and now holds the result of Substitute Into */
str4 = Substitute Into( str2, "All", "Some" );
Show( str1, str2, str3, str4 );
```

This script returns the following output:

```
str1 = "All things considered";
str2 = "Some things considered";
str3 = "Some things considered";
str4 = .;
```

Regular Expressions

A regular expression is a specification of a pattern frequently used to clean up or extract pieces of data. You can search for a pattern and replace it with a different string or extract specific parts of the string. Define the pattern in the Regex() or Regex Match() function.

- "Regex"
- "Regex Match"
- "Special Characters in Regular Expressions"
- "Escaped Characters in Regular Expressions"
- "Greedy and Reluctant Regular Expressions"
- "Backreferences and Capturing Groups"
- "Lookaround Assertions"

Regex

Regex() searches for a *pattern* within a *source* string and returns a string. It simply identifies a pattern in a string or transforms a string into another string.

```
Regex(source, pattern, (<replacement string>, <GLOBALREPLACE>), <format>,
    <IGNORECASE>);
```

IGNORECASE disregards case. GLOBALREPLACE repeats the match until the entire string is processed. format is a backreference to the matched group. Regex() returns missing if the match fails.

Example of Matching a String

bus|car is the regular expression (in quotation marks because it is a string). The expression means match "bus" or "car".

```
sentence = "I took the bus to work.";
vehicle = Regex( sentence, "bus|car" );
    "bus"
```

Examples of Replacing a String

The third optional argument in Regex() is a specification of the result string. The default value, \0, is a backreference to everything that was matched by the regular expression. In the preceding example, the word "bus" is matched in sentence. The default third argument, \0, replaces the entire sentence with "bus".

A more interesting variation uses parentheses to create additional backreferences.

```
sentence = "I took the bus to work.";
Regex( sentence, "(.*) bus (.*)", "\1 car \2" );
   "I took the car to work."
```

The (.*) before and after bus are part of the regular expression. The parentheses create a capturing group. The . matches any character. The * matches zero or more of the previous expression. As a result, the first parenthesis pair matches everything before bus, and the second parenthesis pair matches everything after bus. The third argument, \1 car \2, reassembles the text; it leaves out bus and substitutes car.

See "Backreferences and Capturing Groups" on page 181.

Example of Global Replacement

GLOBALREPLACE changes the behavior of Regex(). If the match succeeds, the entire source string is returned with substitutions made for each place where the pattern matches. If there are no matches, an unchanged source string is returned.

```
sentence = "I took the red bus followed by the blue bus to get to work
   today.";
Regex( sentence, "bus", "car", GLOBALREPLACE);
   "I took the red car followed by the blue car to get to work today."
```

You can also use backreferences. This example starts with a different sentence.

```
sentence = "I took the red bus followed by the blue car to get to work
   today.";
Regex(
   sentence,
   "(\w*) (bus|car)",
   "bicycle (not \2) that was \1",
   GLOBALREPLACE
);
   "I took the bicycle (not bus) that was red followed by the bicycle (not car)
      that was blue to get to work today."
```

The \w* matches zero or more word characters and becomes backreference 1 because of the parentheses. bus|car becomes backreference 2 because of the parentheses. The third argument, bicycle (not \2) that was \1, describes how to build the substitution text for the part of the source text that was matched.

Notice how the backreferences can be used to swap data positions. This might be useful for swapping the position of first names and last names.

Regex Match

`Regex Match()` returns an empty list with zero elements if the match fails. If the match succeeds, the first list is the text of the entire match (backreference 0). The second list is the text that matches backreference 1, and so on.

```
Regex Match(source, pattern, <replacement>|<MATCHCASE>, <NULL>)
```

Unlike `Regex()`, `Regex Match()` is case insensitive. Include `MATCHCASE` for a case-sensitive match. Include `NULL` if you want to match case but there is no replacement text.

Example of Parsing Name-Value Pairs

The following example parses pairs of names and values.

```
Regex Match(
    "person=Fred id=77 friend= favorite=tea",
    "(\w+)=(\S*) (\w+)=(\S*) (\w+)=(\S*) (\w+)=(\S*)"
);
    {"person=Fred id=77 friend= favorite=tea", "person", "Fred", "id", "77",
        "friend", "", "favorite", "tea"}
```

The \w+ matches one or more word characters. The \S* matches zero or more characters that are not spaces. In the resulting JSL list, the field names (`person, id, friend, favorite`) and their corresponding values (`Fred, 77, "", tea`) are separate strings.

If the first argument to `Regex Match()` is a variable and a third argument specifies the replacement value, the matched text is replaced in the variable.

Comparing Regex and Regex Match

`Regex()` and `Regex Match()` match a pattern in a given string but return different results. To transforms your string into another string, use `Regex()`. To identify the substrings that match specific parts of the pattern, use `Regex Match()`.

This example shows the efficiency of `Regex Match()` compared to `Regex()`. The `source` is a list of six strings. The goal is to extract portions of those six strings into the `subject`, `verb`, and `object` columns of a data table.

Figure 6.4 Final Data Table

	subject	verb	object
1	cat	ate	chicken
2	dog	chased	cat
3	ralph	like	mary
4	girl	pets	dog
5	cat	chased	dog

```
source = {"the cat ate the chicken", "the dog chased the cat", "did ralph like
    mary", "the girl pets the dog",
"these words are strange", "the cat was chased by the dog"};

dt = New Table( "English 101", // create the data table
    New Column( "subject", character ),
    New Column( "verb", character ),
    New Column( "object", character )
);
// iterate through the strings in the list
For( i = 1, i <= N Items( source ), i++,

    // assign the result of each match to matchList
    matchList = Regex Match(
        source[i],

    /* scan each string, match zero or more characters
    and one item in each group */

    ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)
    "
    );

    /* If matchList has zero items (string 5), don't add a row
    to the table. Put each matched string in separate
    data table cells. */
    If( N Items( matchList ) > 0,
        dt << Add Rows( 1 );
        dt:subject = matchList[2]; // match the first open parenthesis
        dt:verb = matchList[3]; // match the second open parenthesis
        dt:object = matchList[4]; // match the third open parenthesis
    );
);
```

Regex Match() returns {"the cat was chased by the dog", "cat", "chased", "dog"}
in a single try with each answer in a separate string. Compare this example to a similar one
using Regex(), which returns one answer at a time and builds the final string using
backreferences.

```
For( i = 1, i <= N Items( source ), i++,
    s = Regex( source[i],
    ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)
    ", "\1" ); // match an item in the first group
    v = Regex( source[i],
    ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)
    ", "\2" ); // match an item in the second group
```

```
      o = Regex( source[i],
      ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)
      ", "\3" ); // match an item in the third group
      If( !Is Missing( s ) & !Is Missing( v ) & !Is Missing( o ),
         dt << Add Rows( 1 );
         dt:subject = s; // return the match for \1
         dt:verb = v; // return the match for \2
         dt:object = o; // return the match for \3
      );
   );
```

Backreferences are discussed in "Backreferences and Capturing Groups" on page 181.

Special Characters in Regular Expressions

Special characters are commonly used in regular expressions. The period is a special character that matches one instance of the specified character. It must be escaped with a backslash to be interpreted as a period. In the following expression, the period is replaced with an exclamation point.

```
Regex( "Bicycling makes traveling to work fun.", "\.", "!", GLOBALREPLACE );
   "Bicycling makes traveling to work fun!"
```

Table 6.9 describes the special characters and provides examples.

Table 6.9 Special Characters in Regular Expressions

\	• Precedes a literal character. <\/a> interprets the forward slash literally in the end HTML anchor tag. • Precedes an escape sequence. \n matches a newline character.
^	Matches the beginning of a string, not including the newline character. ^apple matches "apple" at the beginning of a string.
$	Matches the end of a string, not including the newline character. apple$ matches "apple" at the end of a string.
.	Matches any single character including a newline character. .apple matches any single character and then "apple".
|	Represents a logical OR to separate alternative values. (apple|orange|banana) matches "apple", "orange", or "banana".

Table 6.9 Special Characters in Regular Expressions *(Continued)*

?	Matches zero or one instance.
	apple (pie)? matches one or more instances of "pie".
*	Matches zero or more instances.
+	Matches one or more instances.
( )	Encloses a sub-expression.
	(apple|orange|banana) matches "apple", "orange", or "banana".
	^(\w+) matches the beginning of a line and then one or more word characters.
[ ]	Encloses an expression that matches set of characters.
	[\s] matches a whitespace character or a digit.
	[a-z0-9] matches "a" through "z" and numbers "0" through "9".
{ }	Encloses an expression that represents repetition.
	apple{3} repeats three times.
	apple{3,} repeats at least three times as many times as possible.
	apple{3, 10} repeats three times but no more than 10 times.
	Append a question mark to indicate repeating as few times as possible. For example, apple{3,}? repeats at least three times as few times as possible.

Escaped Characters in Regular Expressions

The backslash in a regular expression precedes a literal character. You also escape certain letters that represent common character classes, such as \w for a word character or \s for a space. The following example matches word characters (alphanumeric and underscores) and spaces.

```
Regex(
   "Are you there, Alice?, asked Jerry.", // source
   "(here|there).+(\w+).+(said|asked)(\s)(\w+)\." ); // regular expression
   "there, Alice?, asked Jerry."
```

(here|there).+	Matches "there", a comma, and a space.
(\w+)	Matches "Alice".

.+	Matches "?, ".
(said|asked)(\s)	Matches "asked" followed by a space. Without the space, the match would end here; "asked" is followed by a space in the source string.
(\w+)\.	Matches "Jerry" and a period.

Table 6.10 describes the escaped characters supported in JMP. \C, \G, \X, and \z are not supported.

Table 6.10 Escaped Characters

\\	single backslash
\A	start of a string
\b	word boundary. The zero-length string between \w and \W or \W and \w.
\B	not at a word boundary
\c*X*	ASCII control character
\d	single digit [0-9]
\D	single character that is NOT a digit [^0-9]
\E	stop processing escaped characters
\l	match a single lowercase letter [a-z]
\L	single character that is not lowercase [^a-z]
\Q	ignore escaped characters until \E is found
\r	carriage return
\s	single whitespace character
\S	single character that is NOT white space
\u	single uppercase character [A-Z]
\U	single character that is not uppercase [^A-Z]
\w	word character [a-zA-Z0-9_]
\W	single character that is NOT a word character [^a-zA-Z0-9_]

Table 6.10 Escaped Characters *(Continued)*

\x*00*-\x*FF*	hexadecimal character
\x{*0000*}-\x{*FFFF*}	Unicode code point
\Z	end of a string before the line break

Greedy and Reluctant Regular Expressions

The ?, *, and + operators are greedy by default. They match as many of the preceding character as possible. The ? operator makes them reluctant; ?? matches 0, then 1 if needed; +? matches 1 and then additional characters; *? matches 0 and then additional characters.

The following example starts at the letter n and compares it to the first \d (digits) in the pattern. No digit matches. Because the pattern does not begin with ^ (start of line), the matcher advances to u. The process repeats until the 3 matches the first \d and the 2 matches the second \d.

```
Regex( "number=32.5", "\d\d" );
   "32"
```

Change the pattern to use the greedy + (match one or more).

```
Regex( "number=324.5", "\d+" );
   "324"
```

The preceding example begins much the same, but as soon as the 3 is found and the \d matches, the + greedily matches the 2 and the 4.

Usually, the greedy behavior makes pattern matching faster because the string is consumed sooner. Sometimes a reluctant behavior is better. Adding the ? after the * or + changes them from greedy to reluctant.

```
Regex( "number=324.5", "\d+?" );
   "3"
```

Here, + requires at least one match of a digit character, but ? changes it from "as many as possible" to "as few as possible". It stopped after the 3 because the pattern was satisfied.

Compare the following results.

Greedy:

```
Regex( "number=324.5", "(\d+)(\d+)\.", "first=\1 second=\2" );
   "first=32 second=4"
```

Reluctant:

```
Regex( "number=324.5", "(\d+?)(\d+?)\.", "first=\1 second=\2" );
   "first=3 second=24"
```

In the greedy example above, the matcher greedily matched 3, 2, and 4 for the first \d+. The matcher then had to give back the 4 so that the second \d+ could match something. The reluctant example followed a different path to get a different answer. Initially, the second value was 2, but the pattern could not match the period to the 4, so the second \d+? reluctantly matched the 4 as well.

Use the Reluctant Match for Speed

The greedy and reluctant matches usually produce the same result but not always. See the previous section. One reason you might need the reluctant match is for speed. Suppose that you have a million-character string that begins "The quick fox…" and you want to find the word before "fox". You might write the following expression and expect \1 to contain "quick".

```
The (.+) fox
```

\1 might contain "quick" eventually, after the .+ grabs the million characters to the end of the string and then gives them up, one at a time, until "fox" is found. If there is more than one "fox", it will be the last fox, not this one. To speed it up and make sure we get the first fox, add the ? operator.

```
The (.+?) fox
```

The ? advances one character at a time to get past "quick" and find the first "fox". This method is much faster than going too far.

Typically, the + or * operator is applied to a more restrictive expression such as \d* to match a run of digits, and greedy is faster than reluctant.

Aside from the multiple fox possibility, greedy and reluctant eventually get the same answer. Using the right operator speeds up the match. The right one might be greedy, or it might be reluctant. It depends on what is being matched.

The greedy .* finds the last fox after backing up.

```
Regex(
  "The quick fox saw another fox eating grapes",
  "The (.*) fox",
  "\1"
);
  "quick fox saw another"
```

The reluctant .*? stops on the first fox.

```
Regex(
  "The quick fox saw another fox eating grapes",
  "The (.*?) fox",
  "\1"
);
  "quick"
```

The greedy `.*` has to back up a lot. There is no second fox.

```
Regex(
    "The quick fox saw another animal eating grapes",
    "The (.*) fox",
    "\1"
);
    "quick"
```

The greedy word character match is an even better choice for this problem.

```
Regex(
    "The quick fox saw another fox eating grapes",
    "The (\w*) fox",
    "\1"
);
    "quick"
```

Backreferences and Capturing Groups

A regular expression can consist of patterns grouped in parentheses, also known as capturing groups. In `([a-zA-Z])\s([0-9])`, `([a-zA-Z])` is the first capturing group; `([0-9])` is the second capturing group.

Use a backreference to replace the pattern matched by a capturing group. In Perl, these groups are represented by the special variables $1, $2, $3, and so on. ($1 indicates text matched by the first parenthetical group.) In JMP, use a backslash followed by the group number (\1, \2, \3).

The following example includes a third argument that specifies the replacement text and backreferences.

```
Regex(
    "   Are you there, Alice?, asked Jerry.", // source
    " (here|there).+ (\w+).+(said|asked) (\w+)\.", // regular expression
"  I am \1, \4, replied \2." ); // optional format argument
    "  I am there, Jerry, replied Alice."
```

" I am \1,	Creates the text "I am", a space, and then the first matched pattern, "there".
\4,	Creates the text "Jerry" with the fourth matched pattern (\w+).
replied \2."	Creates the text "replied" and a space. Matches "Alice." with the second matched pattern (\w+).

Lookaround Assertions

Lookaround assertions check for a pattern but do not return that pattern in the results. *Lookaheads* look forward for a pattern. *Lookbehinds* look back for a pattern.

Negative Lookahead Example

Negative lookaheads check for the absence of a pattern before a specific pattern. ?! indicates a negative lookahead. The following expression matches a comma *not* followed by a number or space and replaces the pattern with a comma and space:

```
Regex( "one,two 1,234 cat,dog,duck fish, and chips, to go",
   ",(?!\d|\s)", ", ", GLOBALREPLACE );
   "one, two 1,234 cat, dog, duck fish, and chips, to go"
```

Positive Lookahead Example

Positive lookaheads check for the presence of a pattern before a specific pattern. ?= indicates a positive lookahead. The following expression has the same result as the preceding negative lookahead but matches a comma followed by any lowercase character:

```
Regex( "one,two 1,234 cat,dog,duck fish, and chips, to go",
   ",(?=[a-z])", ", ",GLOBALREPLACE );
   "one, two 1,234 cat, dog, duck fish, and chips, to go"
```

Positive Lookbehind Example

In this example, the positive lookbehind regular expression matches the "ssn=" or "salary=" keywords without including the keyword in the matched text. The matched text is the string of characters that consists of zero or more dollar signs, digits, and hyphens.

```
data = "name=bill salary=$5 ssn=123-45-6789 age=13,name=mary salary=$6
   ssn=987-65-4321 age=14";
redacted = Regex(data, "(?<=(ssn=)|(salary=))[$\d-]*", "###", GLOBALREPLACE);
   "name=bill salary=### ssn=### age=13,name=mary salary=### ssn=### age=14"
```

Here is another way to get the same result using a backreference substitution. ((ssn=)|(salary=)) is the capturing group. "\1" is the backreference to that group.

```
data = "name=bill salary=$5 ssn=123-45-6789 age=13,name=mary salary=$6
   ssn=987-65-4321 age=14";
redacted = Regex(data, "((ssn=)|(salary=))[$\d-]*", "\1###", GLOBALREPLACE);
   "name=bill salary=### ssn=### age=13,name=mary salary=### ssn=### age=14"
```

Backreferences are discussed in "Backreferences and Capturing Groups" on page 181.

Pattern Matching

Pattern matching in JSL is a flexible method for searching and manipulating strings.

You define and use pattern variables just like any JMP variable:

```
i = 3; // a numeric variable
a = "Ralph"; // a character variable
t = textbox("Madge"); // a display box variable
p = ( "this" | "that" ) + patSpan(" ")
      + ( "car" | "bus" ); // a pattern variable
```

When the above statement executes, p is assigned a pattern value. The pattern value can be used either to construct another pattern or to perform a pattern match. The patSpan function returns a pattern that matches a span of characters specified in the argument; patSpan("0123456789") matches runs of digits.

```
p2 = "Take " + p + "."; // using p to build another pattern
If( Pat Match( "Take this bus.", p2 ), // performing a match
  Print( "matches" ),
  Print( "no match" )
);
```

Sometime all you need to know is that the pattern matched the source text, as above. Other times, you might want to know what matched; for example, was it a bus or a car?

```
p = ("this" | "that") + Pat Span( " " ) + ("car" | "bus") >?
vehicleType; // conditional assignment ONLY if pattern matches
If( Pat Match( "Take this bus.", p ),
  // do not use vehicleType in the ELSE because it is not set
  Show( vehicleType ),
  Print( "no match" )
);
```

You could pre-load vehicleType with a default value if you do not want to check the outcome of the match with an if. The >? conditional assignment operator has two arguments, the first being a pattern and the second a JSL variable. >? constructs a pattern that matches the pattern (first argument) and stores the result of the match in the JSL variable (second argument) after the pattern succeeds. Similarly, >> does not wait for the pattern to succeed. As soon (and as often) as the >> pattern matches, the assignment is performed.

```
findDelimString = Pat Len( 3 ) >> beginDelim + Pat Arb() >? middlePart
+Expr( beginDelim );
testString = "SomeoneSawTheQuickBrownFoxJumpOverTheLazyDog'sBack";
rc = Pat Match( testString, findDelimString, "<<<" || middlePart || ">>>" );
Show( rc, beginDelim, middlePart, testString );
```

The above example shows a third argument in the `patMatch` function: the replacement string. In this case, the replacement is formed from a concatenation (|| operator) of three strings. One of the three strings, `middlePart`, was extracted from the `testString` by >? because the replacement cannot occur unless the pattern match succeeds (`rc == 1`).

Look at the pattern assigned to `findDelimString`. It is a concatenation of 3 patterns. The first is a >> operator that matches 3 characters and assigns them to `beginDelim`. The second is a >? operator that matches an arbitrary number of characters and, when the entire match succeeds, assigns them to `middlePart`. The last is an unevaluated expression, consisting of whatever string is in `beginDelim` at the time the pattern is executing, *not* at the time the pattern is built. Just like `expr()`, the evaluation of its argument is postponed. That makes the pattern hunt for two identical three letter delimiters of the middle part.

Other pattern functions might be faster and represent the problem that you are trying to solve better than writing a lot of alternatives; for example, `"a"|"b"|"c"` is the same as `Pat Any("abc")`. The equivalent example for `Pat Not Any("abc")` is much harder. Similar to `Pat Span` (above), `Pat Break("0123456789")` matches up to, but not including, the first number.

Here is a pattern that matches numbers with decimals and exponents and signs. It also matches some degenerate cases with no digits; look at the pattern assigned to digits.

```
digits = Pat Span( "0123456789" ) | "";

number = (Pat Any( "+-" ) | "") >? signPart + (digits) >? wholePart + ("."
+digits | "") >? fractionPart + (Pat Any( "eEdD" ) + (Pat Any( "+-" ) | "")
 + digits | "") >? exponentPart;

If( Pat Match( "-123.456e-78", number ),
   Show( signPart, wholePart, fractionPart, exponentPart )
);
```

Parsing Strings in Fixed Fields

Sometimes data is in fixed fields. The `Pat Tab()`, `Pat R Tab()`, `Pat Len()`, `Pat Pos()`, and `Pat R Pos()` functions make it easy to split out the fields in a fixed field string. `Pat Tab()` and `Pat R Tab()` work from the left and right end of the string and take a number as their argument. They succeed by matching forward to the specified tab position. For example:

```
p = Pat Pos(10) + Pat Tab(15);
```

`Pat Pos(10)` matches the null string if it is in position 10. So at match time, the matcher works its way forward to position 10, then `Pat Tab(15)` matches text from the current position (10) forward to position 15. This pattern is equivalent to `Pat Pos(10)+patLen(5)`. Another example:

```
p = Pat Pos(0) + Pat R Tab(0);
```

This example matches the entire string, from 0 characters from the start to 0 characters from the end. the `Pat Rem()` function takes no argument and is shorthand for `Pat R Tab(0)`; it means the remainder of the string. Pattern matching can also be anchored to the beginning of the string like this:

```
Pat Match( "now is the time", Pat Len(15) + Pat R Pos(0), NULL, ANCHOR );
```

The above pattern uses NULL rather than a replacement value, and ANCHOR as an option. Both are uppercase, as shown. NULL means that no replacement is done. ANCHOR means that the match is anchored to the beginning of the string. The default value is UNANCHORED.

Patterns can be built up like this, but this is *not* recursive:

```
p = "a" | "b"; // matches one character
p = p + p; // two characters
p = p + p; // four characters
Pat Match( "babb", Pat Pos(0) + p + Pat R Pos(0) );
```

A recursive pattern refers to its current definition using Expr():

```
p = "<" + Expr(p) + "*" + Expr(p) + ">" | "x";
Pat Match( "<<x*<x*x>>*x>", Pat Pos(0) + p + Pat R Pos(0) );
```

Remember, `expr()` is the procrastination function; when the pattern is assigned to the variable `p`, `expr()` delays evaluating its argument (`p`) until later. In the next statement, `patMatch` performs the pattern match operation, and each time it encounters `expr()`, it looks for the current value of the argument. In this example, the value does not change during the match). So, if `p` is defined in terms of itself, how can this possibly work?

`p` consists of two alternatives. The right hand choice is easy: a single letter x. The left side is harder: <p*p> . Each `p` could be a single letter x, since that is one of the choices `p` could match, or it could be <p*p>. The last few example have used `patPos(0) + ... + patRPos(0)` to make sure the pattern matches the entire source text. Sometimes this is what you want, and sometimes you would rather the pattern match a subtext. If you are experimenting with these examples by changing the source text, you probably want to match the entire string to easily tell what was matched. The result from `Pat Match` is 0 or 1.

This example uses "Left" recursion:

```
x =  Expr(x) + "a" | "b"; // + binds tighter than |
```

If the pattern is used in FULLSCAN mode, it eventually uses up all memory as it expands. By default, the `patMatch` function does not use FULLSCAN, and makes some assumptions that allow the recursion to stop and the match to succeed. The pattern matches either a "b", or anything the pattern matches followed by an "a".

```
rc = Pat Match( "baaaaa", x );
```

Patterns and Case

Unlike regular expressions, pattern matching is case insensitive. To force case sensitivity, you can add the named argument MATCHCASE to either Pat Match() or Regex Match(). For example:

```
string = "abcABC";

result = Regex Match( string, Pat Regex( "[aBc]+" ) );
Show( string, result );
    / *returns string = "abcABC"
  result = {"abcABC"}*/

result = Regex Match( string, Pat Regex( "[cba]+" ), NULL, MATCHCASE );
Show( string, result );
    /* returns string = "abcABC"
  result = {"abc"}*/
```

Troubleshooting Patterns

When a pattern isn't working the way you expect, use Log() to send pieces of the match to the JMP log. The following example shows how to debug pattern matching and rewrite a script to account for the white space between words in the phrase "the quick brown fox".

```
// greedy match for one or more characters except for white space
word = Pat Repeat( Pat Not Any( " " ) );

Pat Match(
  "the quick brown fox",
  word >> a +
  word >> b +
  word >> c +
  word >> d
);

Show( a, b, c, d );
  a = "qu";
  b = "i";
  c = "c";
  d = "k";
```

The output is probably not what you want, given the name of the pattern *word*. None of the words in the pattern were matched.

The Show(a, b, c, d) expression is a good first step. If it doesn't tell you how JMP got that answer, try inserting Log():

```
word = Pat Repeat( Pat Not Any( " " ) );
```

```
Pat Match(
  "the quick brown fox",
  (word >> a) >> Log( "a" ) +
  (word >> b) >> Log( "b" ) +
  (word >> c) >> Log( "c" ) +
  (word >> d) >> Log( "d" )
);

Show( a, b, c, d );
  0(a) the (the number 0 is the position of the match)

  0(a) th (the "a" is an aribtrary identifier you supplied to the log)

  2(b) e

  ...
  4(a) quick
  4(a) quic
  ...
```

With this extra information, you can now see into the workings of the match, how the word pattern does, in fact, match a word, at least initially. But then it backs up and tries a smaller amount of the word so that the next word pattern can match some non-whitespace characters. However, there's no provision in this pattern to match the white space between words. And that's exactly what it did (find four consecutive words of one or more non-whitespace characters). Maybe the following script is what you need:

```
word = Pat Repeat( Pat Not Any( " " ) );
gap = Pat Repeat( Pat Any( " " ), 1, 999, GREEDY );

Pat Match(
  "the quick brown fox",
  word >> a + gap +
  word >> b + gap +
  word >> c + gap +
  word >> d
);

Show( a, b, c, d );
  a = "the";
  b = "quick";
  c = "brown";
  d = "fox";
```

Data Structures

Working with Collections of Data

JSL provides these basic data structures that can hold a variety of data in a single variable:

- A list holds a number of other values, including nested lists and expressions.

- A matrix is a row-by-column table of numbers.

- An associative array maps keys to values, which can be almost any other type of data.

Contents

Lists

Lists are containers to store items, such as the following:

- numbers
- variables
- character strings
- expressions (for example, assignments or function calls)
- matrices
- nested lists

Create a list in one of the following ways:

- use the List function
- use { } curly braces

Examples

Use the List() function or curly braces to create a list that includes numbers and variables:

```
x = List(1, 2, b);
x = {1, 2, b};
```

A list can contain text strings, nested lists, and function calls:

```
{"Red", "Green", "Blue", {1, "true"}, sqrt( 2 )};
```

You can place a variable into a list and assign it a value at the same time:

```
x = {a = 1, b = 2};
```

Evaluate Lists

When you run a script that contains a list, a copy of the list is returned. The items inside the list are not evaluated.

```
b = 7;
x = {1, 2, b, Sqrt( 3 )};
Show( x );
    x = {1, 2, b, Sqrt(3)};
```

To evaluate items in a list, use the Eval List() function.

```
b = 7;
x = {1, 2, b, Sqrt( 3 )};
c = Eval List( x );
    {1, 2, 7, 1.73205080756888}
```

When using a list of variables that reference lists, you will need to use an Eval() function.

Consider the following example, where you have a variable called fullMonth that is a list of 12 items.

```
::fullMonth = {January, February, March, April, May, June, July, August,
  September, October, November, December};
::abbrevMonth = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
::dow = {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
::levels = {Very Low, Low, Medium Low, Medium, Medium High, High, Very High};
::feel = {Strongly Disagree, Disagree, Neutral, Indifferent, Agree, Strongly
  Agree};
::rating = {Failing, Unacceptable, Very Poor, Poor, Bad, Acceptable, Average,
  Good, Better, Very Good, Excellent, Best};
::mlist = {::fullMonth, ::abbrevMonth, ::dow, ::levels, ::feel, ::rating};
N Items(::fullMonth); // returns 12 because fullMonth has 12 items.
N Items(::mlist[1]); /* returns this message: "N Items() argument must be a
  list" because ::mlist[1] is a variable (fullMonth) and fullMonth is a
  list.*/
```

When you add the Eval() function:

```
N Items( Eval( ::mlist[1] ) );
  12
/* returns 12 because::mlist[1] holds ::fullMonth, and evaluating it
returns ::fullMonth's list.*/
```

In a loop, you need the Eval() function in the nested loop to evaluate the variable's contents. For example:

```
For( g = 1, g <= N Items( ::mlist ), g++,
    For( i = 1, i <= N Items(::mlist[g]), i++,
        Show( ::mlist[g][i] )
    )
);
// returns the message "N Items() argument must be a list."
```

To fix the problem, add the Eval() function:

```
::mlist = {::fullMonth, ::abbrevMonth, ::dow, ::levels, ::feel, ::rating};
For( g = 1, g <= N Items( ::mlist ), g++,
    For( i = 1, i <= N Items( Eval( ::mlist[g] ) ), i++,
        Show( ::mlist[g][i] )
    )
);
```

Assignments with Lists

Create a list to assign values to variables.

Examples

```
{a, b, c} = {1, 2, 3}; // assign 1 to a, 2 to b, and 3 to c
{a, b, c}--; // decrement a, b, and c
{{a}, {b, c}}++; // increment a, b, and c by 1
mylist = {1, log( 2 ), e()^pi(), height[40]}; // store the expressions
```

Perform Operations in Lists

In lists, you can perform operations.

```
a = {{1, 2}, 3, {4, 5}};
b = {{10, 20}, 30, {40, 50}};
c = a + b;
   c = {{11, 22}, 33, {44, 55}}
```

Find the Number of Items in a List

To determine the number of items in a list, use the N Items() function.

```
x = {1, 2, y, Sqrt( 3 ), {a, b, 3}};
N = N Items( x );
Show( n );
   n = 5;
```

Subscripts

Subscripts extract specified items from a list. Use a list as a subscript to return multiple items from a list.

Note: JSL starts counting from 1, so the first element in a list is [1], not [0] as in some other languages.

Examples

List a contains four items.

```
a = {"bob", 4, [1, 2, 3], {x, y, z}};
Show( a[1] );
   a[1] = "bob";
Show( a[{1, 3}] );
```

```
   a[{1, 3}] = {"bob", [1, 2, 3]};
a[2] = 5; // assign 5 to the second list item
```

You can also use subscripts to select or change items in a list:

```
Show( a );
   a = {"bob", 5, [1, 2, 3], {x, y, z}};
c = {1, 2, 3};
c[{1, 2}] = {4, 4};
// c[{1, 2}] = 4 produces the same result
Show( c );
   c = {4, 4, 3};
```

When you have assignments or functions in a list, you can use a quoted name for the subscript to extract the value.

```
x={sqrt( 4 ), log( 3 )};
xx= {a = 1, b = 3, c = 5};
x["sqrt"];
   4
xx["b"];
   3
```

The name must be in quotation marks, or else JMP tries to evaluate it and use its value. The following example shows the values of the second item in the list, rather than the value of a in the list.

```
a = 2;
Show( xx[a] );
   xx[a] = b = 3;
```

Notes:

- Multiple left-side subscripts (for example, a[i][j] = value where a contains a list of things that are subscriptable) are allowed in the following circumstances:

  – Each level except the outermost level must be a list. So, in the example above, a must be a list but a[i] can be anything subscriptable.

  – Each subscript except the last must be a number. So, in the example above, i must be a number, but j could be a matrix or list of indices.

- Subscripting can be done to any level of nesting, such as the following:

  a[i][j][k][l][m][n] = 99;

Note: To get a value in the matrix of an expression column, use double subscripts, as in :Discrim Data Matrix[row()][i].

Locate Items in a List

Use the Loc() function or the Contains() function to find values in a list:

```
Loc( list, value );
Contains( list, value );
```

Loc() and Contains() return the positions of the values. Loc() returns the results in a matrix, and Contains() returns the results as a number.

Notes:

• The Loc function returns each occurrence of a repeated value. Contains() returns only the first occurrence of a repeated value.

• If the value is not found, the Loc function returns an empty matrix and Contains() returns a zero.

• To assess whether an item is in a list, use Loc() and Contains() with >0. A returned value of zero means that the item is not in the list. A returned value of 1 means that the item is in the list at least once.

Note: For more information about matrix manipulation and a description of the equivalent Loc() command for matrices, see "Matrices" on page 202.

Examples

```
nameList = {"Katie", "Louise", "Jane", "Jane"};
numList = {2, 4, 6, 8, 8};
```

Search for the value "Katie" in the nameList:

```
Loc( nameList, "Katie" );
  [1]
Contains( nameList, "Katie" );
  1
```

Search for the value "Erin" in the nameList:

```
Loc( nameList, "Erin" );
  []
Contains( nameList, "Erin" );
  0
```

Search for the number 8 in the numList:

```
Loc( numList, 8 );
  [4, 5]
Contains( numList, 8 );
  4
```

Find out if the number 5 exists in the numList:

```
NRow( Loc( numList, 5 ) ) > 0;
   0
Contains( numList, 5 ) > 0;
   0
```

List Operators and Functions

Table 7.1 describes the list operators and their syntax.

Table 7.1 List Operators

Operator and Function		Syntax	Explanation
	As List()	As List(*matrix*)	Returns the matrix as a list. A matrix with multiple columns is returned as a list of lists, one list per row.
= += -= *= /= ++ --	Assign() Add To() SubtractTo() MultiplyTo() DivideTo() Post Increment() Post Decrement()	{*list*} = {*list*} {*list*} += *value* {*list*} -= {*list*} ...	If the target of an assignment operator is a list and the value to be assigned is a list, then it assigns item by item. The ultimate values in the left list must be L-values (in other words, names capable of being assigned values). **Notes:** • If you want to test equality of lists, use ==, not =. • JMP does not have pre-increment or pre-decrement operators. Instead, use the Add To() operator, which is (+=), or the Subtract To() operator, which is (-=).
||=	Concat To()	Concat To(*list1*, *list2*, ...)	Inserts the second and subsequent lists at the end of the first list.
||	Concat()	Concat(*list1*, *list2*, ...);	Returns a copy of the first list with any additional lists inserted at after it.

Table 7.1 List Operators *(Continued)*

Operator and Function	Syntax	Explanation
Eval List()	Eval List(*list*)	Returns a list of the evaluated expressions inside list. See "Evaluate Lists" on page 191.
Insert Into()	Insert Into(*list*, *x*, <*i*>)	Inserts a new item (*x*) into the *list* at the given position (*i*). If *i* is not given, the item is added to the end of the list. This function does change the original list.
Insert()	*list* = Insert(*list*, *x*, <*i*>)	Returns a copy of the *list* with a new item (*x*) inserted into the *list* at the given position (*i*). If *i* is not given, the item is added to the end of the list. This function does not change the original list.
Is List()	Is List(*arg*)	Returns true (1) if *arg* is a classical list (in other words, one that would result from the construction by List(*items*) or {*items*}) and returns false (0) otherwise. An empty list is still a list, so IsList({ }) returns true. If miss=., then IsList(miss) returns false, not missing.
{ } List	List(*a*, *b*, *c*) {*a*, *b*, *c*}	Constructs a list from a set of items. An item can be any expression, including other lists. Items must be separated by commas. Text should either be enclosed in double quotation marks (" ") or stored in a variable and called as that variable.
N Items	N Items(*list*)	Returns the number of elements in the *list* specified. Can be assigned to a variable.

Table 7.1 List Operators *(Continued)*

Operator and Function	Syntax	Explanation
Remove From()	Remove From(*list*, <*i*>, <*n*>)	Deletes *n* items from the *list*, starting from the indicated position (*i*). If *n* is omitted, the item at *i* is deleted. If *n* and *i* are omitted, the item at the end is removed. This function does change the original list.
Remove()	Remove(*list*, <*i*>, <*n*>)	Returns a copy of the *list* with the *n* items deleted, starting from the indicated position (*i*). If *n* is omitted, the item at *i* is deleted. If *n* and *i* are omitted, the item at the end is removed. This function does not change the original list.
Reverse Into()	Reverse Into(*list*)	Reverses the order of the items in the *list*. This function does change the original list.
Reverse()	Reverse(*list*)	Returns a copy of the *list* with the items in reverse order. This function does not change the original list.
Shift Into()	Shift Into(*list*, <*n*>)	Shifts *n* items from the front of the *list* to the end of the *list*. If *n* is omitted, the first item is moved to the end of the list. This function does change the original list.
Shift()	Shift(*list*, <*n*>)	Returns a copy of the *list* with *n* items shifted from the front of the list to the end of the *list*. If *n* is omitted, the first item is moved to the end of the list. This function does not change the original list.
Sort Ascending()	Sort Ascending(*list*)	Returns a copy of the *list* sorted in ascending order. This function does not change the original list.

Table 7.1 List Operators *(Continued)*

	Operator and Function	Syntax	Explanation
	Sort Descending()	Sort Descending(*list*)	Returns a copy of the *list* sorted in descending order. This function does not change the original list.
	Sort List Into()	Sort List Into(*list*)	Sorts the *list* in ascending order. This function does change the original list.
	Sort List()	Sort List(*list*)	Returns a copy of the *list* sorted in ascending order. This function does not change the original list.
[ ]	Subscript()	*list*[*i*] *x* = *list*[*i*] *list*[*i*] = *value* *a*[*b*, *c*] Subscript(*a*, *b*, *c*)	Subscripts for lists extract the ith item from the *list*. Subscripts can in turn be lists or matrices.
	Substitute()	Substitute(*list*, *pattExpr1*, *replExpr1*, ...)	Returns a copy of a string, list, or expression, replacing instances of each pattern expression with the corresponding replacement expression. See "Substitute and Substitute Into" on page 171 in the "Types of Data" chapter.
	Substitute Into()	Substitute Into(*list*, *pattExpr1*, *replExpr1*, ...)	Changes a string, list, or expression, replacing instances of each pattern expression with the corresponding replacement expression. **Note:** The *list* or expression must be a variable. See "Substitute and Substitute Into" on page 171 in the "Types of Data" chapter.

Iterate through a List

Iterate through a list to do something with each value or look for a particular value. The following script looks at each item in the list. If the item in the list is less than or equal to 10, it is replaced with its square.

```
x = {2, 12, 8, 5, 18, 25};
```

```
n = N Items( x );
For( i = 1, i <= n, i++,
  If( x[i] <= 10,
     x[i] = x[i] ^ 2
  )
);
Show( x );
   x = {4, 12, 64, 25, 18, 25};
```

You can use Loc() to locate the items in the new list that are equal to 25:

```
Loc( x, 25 );
   [4, 6] // The fourth and sixth items in the list are equal to 25.
```

Concatenate Lists

Join two or more lists into one list using Concat() or the || operator. The original lists are not changed.

The following example uses Concat() to join lists *a* and *b*:

```
a = {1, 2};
b = {7, 8, 9};
Concat( a, b );
   {1, 2, 7, 8, 9}
```

The following example joins the same lists using the || operator:

```
{1, 2} || {7, 8, 9}
   {1, 2, 7, 8, 9}
```

Lists of different types can be concatenated (for example, lists that contain character strings and numbers).

```
d = {"apples", "bananas"};
e = {"oranges", "grapes"};
f = {1, 2, 3};
Concat( d, e, f);
   {"apples", "bananas", "oranges", "grapes", 1, 2, 3}
```

Join two or more lists and replace the first list with the combined list using Concat to() or the ||= operator. The following example performs concatenation in place using Concat to():

```
d = {"apples", "bananas"};
e = {"peaches", "pears"};
Concat to(d,e);
Show( d );
   d = {"apples", "bananas", "peaches", "pears"}
```

The following example joins the same lists using the ||= operator:

```
d = {"apples", "bananas"};
```

```
e = {"peaches", "pears"};
d||=e;
Show( d );
  d = {"apples", "bananas", "peaches", "pears"}
```

Insert a Nested List in an Existing List

To insert a nested list in an existing list, specify the placement of the list:

```
list2 = {{"a", "b"}, {"c", "d"}};
list2[3] = {"apple", "banana"}; // specify a list as the third item
Show( list2 );
  list2 = {{"a", "b"}, {"c", "d"}, {"apple", "banana"}};
```

If you do not know the number of items in a list, use N Items(). In this example, an empty list is inserted at the end of the list.

```
list1 = {"a", "b"};
list1[N Items( list1 ) + 1] = {};
Show( list1 );
  list1 = {"a", "b", {}};
```

Index into a List with Another List

Indexing into a a list with another list is a powerful technique for making scripts fast and small. See "Indexing into a Matrix or List with Another Matrix or List" on page 213.

Send Messages to Lists

You can send a single message to a list instead of having to send individual messages. Suppose that you want to create a Bivariate report with a By variable. The script returns a list of the bivariate objects, one for each level of the By variable. Send a single Fit Line message to the list to fit a line to all of the Bivariate reports.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
/* return a list of bivariate objects:
{Bivariate[], Bivariate[], Bivariate[], Bivariate[], Bivariate[],
Bivariate[]}; */
biv = dt << Bivariate( X( dt:weight ), Y( dt:height ), By( dt:age ) );

biv << Fit Line; /* send the Fit Line message to
each Bivariate object in the list */
```

Troubleshooting Messages Sent to a List

Any message that is sent to a list is sent to each object in the list. If the message saves a file, it's performing the save for each object in the list to the *same file*. This means that only the last object is actually saved in a file.

Suppose that you have a list of three control charts and you send a message (such as `Save Interactive HTML`) to the list. The message is sent to each member of the list in turn. First, it saves an HTML page with the first control chart; then it saves a second HTML page with the same name that contains the second control chart; and then it saves a third HTML page with the same name that contains only the third control chart. You end up with a page that contains only the third control chart.

The solution is to send the message to a single container higher up the tree that contains all control charts. To do this, rewrite the following line of code:

```
obj << Save Interactive HTML( "Control Charts.html" );
```

`Top Parent` treats the first object as the single container:

```
tp = obj[1] << Top Parent();
tp << Save Interactive HTML( "Control Charts.html" );
```

Matrices

A matrix is a rectangular array of numbers that are arranged in rows and columns. Use matrices to store numbers and perform calculations on those numbers using matrix algebra.

Note the following for this section:

- Matrices are represented with an uppercase bold variable (for example, **A**).
- A matrix with one row or one column is a *vector* (or more specifically, a *row vector* or a *column vector* respectively).
- For clarity, we represent matrices that are vectors with lowercase bold letters (such as **x**).
- A *scalar* is a numeric value that is not in a matrix.

Construct Matrices

Note the following when creating matrices:

- Place matrix literals in square brackets. `[...]`
- Matrix values can contain decimal points, can be positive or negative, and can be in scientific notation.

- Separate items across a column with blank spaces. You can use any number of blank spaces.

- Separate rows with a comma.

For constructing more advanced matrices, see "Special Matrices" on page 219.

Examples

Create matrix **A** with 3 rows and 2 columns:

```
A = [1 2, 3 4, 5 6];
```

R is a row vector and **C** is a column vector:

```
R = [10 12 14];
C = [11, 13, 15];
```

B is a 1-by-1 matrix, or a matrix with one row and one column:

```
B = [20];
```

E is an empty matrix:

```
E = [];
```

The following examples show two ways of creating empty matrices:

```
J( 0,4 );
   [](0,4)
```

```
[]( 5,0 )
   [](5,0)
```

Specifying the number of rows and columns in an empty matrix is optional. JMP creates the matrix as necessary. For example, the following script defines an empty matrix, iterates over each row in the data table, and returns the heights and weights in a matrix.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
data = []( 0, 2 );
For Each Row( data |/= Matrix( {{dt:height, dt:weight}} ) );
Show( data );
   data = [59 95, 61 123, 55 74,...]
```

A script can return an empty matrix. In Big Class.jmp, the following expression looks for rows in which age equals 8, finds none, and returns an empty matrix:

```
a = dt << Get Rows Where( age == 8 );
Show( a );
   a = [](0,1);
```

Construct Matrices from Lists

If you want to convert lists into a matrix, use the `Matrix()` function. A single list is converted into a column vector. Two or more lists are converted into rows.

Create a column vector from a single list:

```
A = Matrix( {1, 2, 3} );
   [1,2,3]
```

Create a matrix from a list of lists. Each list is a row in the matrix.

```
A = Matrix( {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} );
   [1 2 3,
   4 5 6,
   7 8 9]
```

Construct Matrices from Expressions

To construct matrices from expressions, use `Matrix()`. Elements must be expressions that resolve to numbers.

```
A = Matrix( {4 * 5, 8 ^ 2, Sqrt( 9 )} );
   [20, 64, 3]
```

Subscripts

Use the subscript operator (`[ ]`) to pick out elements or submatrices from matrices. The `Subscript()` function is usually written as a bracket notation after the matrix to be subscripted, with arguments for rows and columns.

Single Element

The expression `A[i,j]` extracts the element in row `i`, column `j`, returning a scalar number. The equivalent functional form is `Subscript(A,i,j)`.

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 3]; // returns 6
Subscript( P, 2, 3 ); // returns 6
```

Assign the value that is in the third row and the first column in the matrix **A** (which is 5) to the variable `test`.

```
A = [1 2, 3 4, 5 6];
test = A[3, 1];
Show( test );
   test = 5;
```

Matrix or List Subscripts

To extract a sub-matrix, use matrix or list subscripts. The result is a matrix of the selected rows and columns. The following expressions select the second and third rows with the first and third columns.

```
P = [1 2 3, 4 5 6, 7 8 9];
P[[2 3],[1 3]]; // matrix subscripts
P[{2, 3},{1, 3}]; // list subscripts
```

Both of these methods provide the following output:

```
[4 6,
7 9]
```

Single Subscripts

A single subscript addresses matrices as if all the rows were connected end-to-end in a single row. This makes the double subscript A[i,j] the same as the single subscript A[(i-1)*ncol(A)+j].

Examples

```
Q = [2 4 6, 8 10 12, 14 16 18];
Q[8]; // same as Q[3,2]
   16
```

The following examples all return the column vector [10, 14, 18]:

```
Q = [2 4 6, 8 10 12, 14 16 18];
Q[{5, 7, 9}];
Q[[5 7 9]];
ii = [5 7 9];
Q[ii];
ii = {5, 7, 9};
Q[ii];
Subscript( Q, ii );
```

This script returns the values 1 through 9 from the matrix **P** in order:

```
P = [1 2 3, 4 5 6, 7 8 9];
For( i = 1, i <= 3, i++,
   For( j = 1, j <= 3, j++,
      Show( P[i, j] )
   )
);
```

Delete Rows and Columns

Deleting rows and columns is accomplished by assigning an empty matrix to that row or column.

```
A[k, 0] = [];  // deletes the kth row
A[0, k] = [];  // deletes the kth column
```

Select Whole Rows or Columns

A subscript argument of zero selects all rows or columns.

```
P = [1 2 3, 4 5 6, 7 8 9];
```

Select column 2:

```
P[0, 2];
    [2, 5, 8]
```

Select columns 3 and 2:

```
P[0, [3, 2]];
    [3 2, 6 5, 9 8]
```

Select row 3:

```
P[3, 0];
    [7 8 9]
```

Select rows 2 and 3:

```
P[[2, 3], 0];
    [4 5 6, 7 8 9]
```

Select all columns and rows (the whole matrix):

```
P[0, 0];
    [1 2 3, 4 5 6, 7 8 9]
```

Assignment through Subscripts

You can change values in matrices using subscripts. The subscripts can be single indices, matrices or lists of indices, or the zero index representing all rows or columns. The number of selected rows and columns for the insertion must either match the dimension of the inserted argument, or the argument can be inserted repeatedly into the indexed positions.

Examples

Change the value in row 2, column 3 to 99:

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 3] = 99;
Show( P );
```

P=[1 2 3, 4 5 99, 7 8 9]

Change the values in four locations:

```
P = [1 2 3, 4 5 6, 7 8 9];
P[[1 2], [2 3]] = [66 77, 88 99];
Show( P );
```
  P=[1 66 77, 4 88 99, 7 8 9]

Change three values in one column:

```
P = [1 2 3, 4 5 6, 7 8 9];
P[0, 2] = [11, 22, 33];
Show( P );
```
  P=[1 11 3, 4 22 6, 7 33 9]

Change three values in one row:

```
P = [1 2 3, 4 5 6, 7 8 9];
P[3, 0] = [100 102 104];
Show( P );
```
  P=[1 2 3, 4 5 6, 100 102 104]

Change all the values in one row to the same value:

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 0] = 99;
Show( P );
```
  P=[1 2 3, 99 99 99, 7 8 9]

Operator Assignment

You can use operator assignments (such as +=) on matrices or subscripts of matrices. For example, the following statement adds 1 to the *i* - *j*th element of the matrix:

```
P = [1 2 3, 4 5 6, 7 8 9];
P[1, 1] += 1;
Show( P );
```
  P=[2 2 3,
  4 5 6,
  7 8 9];

```
P[1, 1] += 1;
Show( P );
```
  P=[3 2 3,
  4 5 6,7 8 9];

Ranges of Rows or Columns

If you are working with a range of subscripts, use the Index() function or :: to create matrices of ranges.

```
T1 = 1 :: 3; // create the vector [1 2 3]
T2 = 4 :: 6; // create the vector [4 5 6]
T3 = 7 :: 9; // create the vector [7 8 9]
T = T1 |/ T2 |/ T3; // concatenate the vectors into a single matrix
T[1 :: 3, 2 :: 3]; // refer to rows 1 through 3, columns 2 and 3
    [2 3, 5 6, 8 9]
T[Index( 1, 3 ), Index( 2, 3 )]; // equivalent Index function
    [2 3, 5 6, 8 9]
```

Inquiry Functions

The NCol() and NRow() functions return the number of columns and rows in a matrix (or data table), respectively:

```
NCol( [1 2 3, 4 5 6] ); // returns 3 (for 3 columns)
NRow( [1 2 3, 4 5 6] ); // returns 2 (for 2 rows)
```

To determine whether a value is a matrix, use the Is Matrix() function, which returns a 1 if the argument evaluates to a matrix.

```
A = [20, 64, 3];
B = {20, 64, 3};
Is Matrix( A ); // returns 1 for yes
Is Matrix( B ); // returns 0 for no
```

Comparisons, Range Checks, and Logical Operators

JMP's comparison, range check, and logical operators work with matrices and produce matrices of elementwise Boolean results. You can compare conformable matrices.

```
A < B; // less than
A <= B; // less or equal
A > B; // greater than
A >= B; // greater or equal
A == B; // equal to
A != B; // not equal to
A < B < C; // continued comparison (range check)
A | B; // logical OR
A & B; // logical AND
( A & !B ) | ( !A & B ); // exlusive OR
!( A & B ) & ( A | B ); // exlusive OR
```

You can use the Any() or All() functions to summarize matrix comparison results. Any() returns a 1 if any element is nonzero. All() returns a 1 if all elements are nonzero.

```
[2 2] == [1 2]; // returns [0 1], therefore:
All( [2 2] == [1 2] ); // returns 0
```

```
Any( [2 2] == [1 2] ); // returns 1
```

`Min()` or `Max()` return the minimum or maximum element from the matrix or matrices given as arguments.

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = [0 1 2, 2 1 0, 0 1 1, 2 0 0];
Min( A ); // returns 1
Max( A ); // returns 12
Min( A, B ); // returns 0
```

Numeric Operations

You can perform numeric operations (such as subtraction, addition, and multiplication) on matrices. Most statistical methods are expressed in compact matrix notation and can be implemented in JSL.

For example, the following expression uses matrix multiplication and inversion to illustrate least squares regression:

$$b = (X'X)^{-1}X'y$$

Implement this equation through the following JSL expression:

```
b = Inv( X'*X )*X'*y;
```

Basic Arithmetic

You can perform the following basic arithmetic on matrices:

- addition
- subtraction
- multiplication
- division (multiplying by the inverse)

Note: The standard multiply operator is a matrix multiplier, not an elementwise multiplier.

To perform matrix multiplication, use one of the following methods:

- `*` operator
- `Multiply()` function
- `Matrix Mult()` function

To perform matrix division, use one of the following methods:

- `/` operator

- `Divide()` function

Note the following about matrix multiplication and division:

- Remember that while multiplication or division of scalars is commutative (ab = ba), multiplication or division of matrices is *not* commutative.

- When one of the two elements is a scalar, elementwise multiplication or division is performed.

- To use elementwise multiplication, use `:*` or the `EMult()` function.

- To use elementwise division, use `:/`, or the equivalent `EDiv()` function.

Examples

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = [0 1 2, 2 1 0, 0 1 1, 2 0 0];
C = [1 2 3 4, 4 3 2 1, 0 1 0 1];
D = [0 1 2, 2 1 0, 1 2 0];
```

Matrix addition:

```
R = A + B;
    [1 3 5,
    6 6 6,
    7 9 10,
    12 11 12]
```

Matrix subtraction:

```
R = A - B;
    [1 1 1,
    2 4 6,
    7 7 8,
    8 11 12]
```

Matrix multiplication (inner product of rows of **A** with columns of **C**):

```
R = A * C;
    [9 11 7 9,
    24 29 22 27,
    39 47 37 45,
    54 65 52 63]
```

Matrix division (equivalent to `A*Inverse(D)`):

```
R = A / D;
    [1.5 0.5 0,
    3 2 0,
    4.5 3.5 0,
    6 5 0]
```

Matrix elementwise multiplication:

```
R = A :* B;
```

```
[0 2 6,
8 5 0,
0 8 9,
20 0 0]
```

Matrix scalar multiplication:

```
R = C * 2;
[2 4 6 8,
8 6 4 2,
0 2 0 2]
```

Matrix scalar division:

```
R = C / 2;
[0.5 1 1.5 2,
2 1.5 1 0.5,
0 0.5 0 0.5]
```

Matrix elementwise division (division by zero results in missing values):

```
R = A :/ B;
[. 2 1.5,
2 5 .,
. 8 9,
5 . .]
```

Numeric (Scalar) Functions on Matrices

Numeric functions work elementwise on matrices. Most of the pure numeric functions can be applied to matrices, resulting in a matrix of results. You can mix conformable matrix and scalar arguments.

Examples of numeric functions include the following:

- Sqrt(), Root(), Log(), Exp(), ^ Power(), Log10()
- Abs(), Mod(), Floor(), Ceiling(), Round(), Modulo()
- Sine,(), Cosine(), Tangent(), ArcSine(), and other trigonometry functions.
- Normal Distribution(), and other probability functions.

Example

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = Sqrt( A ); // elementwise square root
[1 1.414213562373095 1.732050807568877,
2 2.23606797749979 2.449489742783178,
2.645751311064591 2.82842712474619 3,
3.16227766016838 3.3166247903554 3.464101615137754]
```

Concatenation

The Concat() function combines two matrices side by side to form a larger matrix. The number of rows must agree. A double vertical bar (||) is the infix operator, equivalent for horizontal concatenation.

```
Identity( 2 ) || J( 2, 3, 4 );
    [1 0 4 4 4, 0 1 4 4 4]

B = [1, 1];
B || Concat( Identity( 2 ), J( 2, 3, 4 ) );
    [1 1 0 4 4 4, 1 0 1 4 4 4]
```

The VConcat() function stacks two matrices on top of each other to form a larger matrix. The number of columns must agree. A vertical-bar-slash (|/) is the infix operator, equivalent for vertical concatenation.

```
Identity( 2 ) |/ J( 3, 2, 1 );
// or VConcat( Identity( 2 ),J( 3, 2, 1 ) );
    [1 0, 0 1, 1 1, 1 1, 1 1]
```

Both Concat() and VConcat() support concatenating to empty matrices, scalars, and lists.

```
a=[];
a || [1]; // returns [1]
a || {2}; // returns [2]
a || [3 4 5]; // returns [3 4 5]
```

There are two in place concatenation operators: ||= and |/= . They are equivalent to the Concat To() and V Concat To() functions, respectively.

- a || = b is equivalent to a = a || b
- a |/ = b is equivalent to a = a | /b

Transpose

The Transpose() function transposes the rows and columns of a matrix. A back-quote (`) is the postfix operator, equivalent to Transpose(). In matrix notation, Transpose() is expressed as the common prime or superscript-T notation (A' or A^T).

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
A`
    [1 4 7 10,
     2 5 8 11,
     3 6 9 12]
Transpose( [1 2, 3 4] );
    [1 3, 2 4]
```

Indexing into a Matrix or List with Another Matrix or List

Indexing into a matrix (or a list) with another matrix (or list) is a powerful technique for making scripts fast and small. JMP 13 and later handles one edge case differently.

Consider the following example of using a matrix as an index into a data column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
tall rows = Loc( :height << Get Values() >= 67 ); // [25, 27, 30, 37, 39, 40]
:height[tall rows]; // [68, 69, 67, 68, 68, 70]
N Rows( :height[tall rows] ); // 6
```

The second line is the key. `:height << Get Values()` returns the values of the `height` column as a matrix. The comparison `>= 67` returns a matrix of 0s and 1s corresponding to where the expression is false (0) or true (1). The `Loc()` function returns the locations of the 1s within the previous matrix.

In this case, you get [25, 27, 30, 37, 39, 40] as the row numbers where the `height` column is greater than or equal to 67. That's pretty handy by itself, but the real power comes from being able to use that matrix as the index into a column or other matrix as in the third and fourth lines.

The alternative to this technique is to write a `For()` loop to iterate through every row explicitly. JMP still loops through the rows in either case, but the looping is much faster if done internally rather than in JSL. And the JSL is more compact without the explicit loop.

You can do the same thing with a matrix variable instead of a data column reference:

```
m = :height << Get Values();
tall rows = Loc( m >= 67 ); // [25, 27, 30, 37, 39, 40]
m[tall rows]; // [68, 69, 67, 68, 68, 70]
N Rows( m[tall rows] ); // 6
```

Consider the same examples with a different height cutoff (70 instead of 67):

```
tall rows = Loc( :height << Get Values() >= 70 ); // [40]
:height[tall rows]; // [70]
N Rows( :height[tall rows] ); // 1

m = :height << Get Values();
tall rows = Loc( m >= 70 ); // [40]
m[tall rows]; // [70]
N Rows( m[tall rows] ); /* error in JMP 12 and prior versions
otherwise, returns 1 */
```

Notice that `m[tall rows]` returned 70 instead of [70]. In many cases, a 1x1 matrix and a number are treated the same way, but not always. The `N Rows()` function is one example where it makes a difference. The indexing is too aggressively simplifying the 1x1 matrix into a number.

One work-around is to call `Is Matrix()` before using `N Rows()`. An alternative is to concatenate an empty matrix to the result, which creates a matrix whether the source is a number or a matrix:

```
m = m |/ []
```

In JMP 13 and later, the 1x1 matrix is maintained so that the result of the index operation is a matrix whenever the index itself is a matrix. The same principle applies to list indexing. Data table column indexing has always followed this principle.

Matrices and Data Tables

You can move information between a matrix and a JMP data table. You can use matrix algebra to perform calculations on numbers that are stored in JMP data tables, and you can save the results back to JMP data tables.

Move Data into a Matrix from a Data Table

These sections describe how to move data from a data table into a matrix.

Move All Numeric Values

The `Get As Matrix()` function generates a matrix containing all of the numeric values in a data table or column:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
A = dt << Get As Matrix;
   [12 59 95,
   12 61 123,
   12 55 74, ...]
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( "Height" );
A = col << Get As Matrix;
   [59, 61, 55, 66, 52, ...]
```

Move All Numeric Values and Character Columns

The `Get All Columns As Matrix()` function returns the values from all columns of the data table in a matrix, including character columns. Character columns are numbered according to the alphanumeric order, starting at 1.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
A = dt << Get All Columns As Matrix;
   [21 12 1 59 95,
   28 12 1 61 123, ...]
```

Move Only Certain Columns

To get certain columns of a data table, use column list arguments (names or characters).

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
x = dt << Get As Matrix( {"height", "weight"} );
```

or

```
x = dt << Get As Matrix( {height, weight} );
  [59 95, 61 123, 55 74, ...]
```

Currently Selected Rows

To get a matrix of the currently selected row numbers in the data table:

```
dt << Get Selected Rows;
```

Note: If no rows are selected, the output is an empty matrix.

Find Rows Where

To see a matrix of row numbers where the expression is true:

```
dt << Get Rows Where( expression );
```

For example, the following script returns the row numbers where the sex is male (M):

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
A = dt << Get Rows Where( Sex == "M" );
  [6, 7, 8, 12, 13, 14, ...]
```

Move Data into a Data Table from a Matrix

This section describes how to move data from a matrix into a data table.

Move a Column Vector

The Set Values() function copies values from a column vector into an existing data table column:

```
col << Set Values( x );
```

col is a reference to the data table column, and **x** is a column vector.

For example, the following script creates a new column called test and copies the values of vector **x** into the test column:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "test" );
col = Column( "test" );
x = 1::40;
```

```
col << Set Values( x );
```

Move All Matrix Values

The Set Matrix() function copies values from a matrix into an existing data table, making new rows and new columns as needed to store the values of the matrix. The new columns are named Col1, Col2, and so on.

```
dt = New Table( "B" );
dt << Set Matrix([1 2 3 4 5, 6 7 8 9 10]);
```

This script creates a new data table called B containing two rows and five columns.

To create a new data table from a matrix argument, use the As Table(matrix) command. The columns are named Col1, Col2, and so on. For example, the following script creates a new data table containing the values of **A**:

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
dt = As Table( A );
```

Matrices and Reports

You can extract matrices of values from reports. First, you need to locate the items that you want to extract. This information is in the tree structure of the report.

Run the following script to create a table of parameter estimates in a Bivariate report:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" ) ;
biv = dt << Bivariate( X( height ), Y( weight ), Fit Line );
```

Now, open the tree structure to identify which item contains the parameter estimates:

• Right-click a gray disclosure icon and select **Edit > Show Tree Structure**.

The parameter estimates are contained in NumberColBox(13). Continue with the script:

```
colBox = Report( biv )[Number Col Box( 13 )];
beta = colBox << Get As Matrix;
  [-127.145248610915, 3.711354893859555]
```

Notes:

• When a variable contains a reference to a table box, Get As Matrix() creates a matrix **A** that contains the values from all numeric columns in the table:

```
A = tableBox << Get As Matrix;
```

• When a variable contains a reference to a numeric column in a report table, Get As Matrix() creates a matrix **A** as a column vector of values from that column.

```
A = colBox << Get As Matrix;
```

Loc Functions

The `Loc()`, `Loc Nonmissing()`, `Loc Min()`, `Loc Max()`, and `Loc Sorted()` functions all return matrices that show positions of certain values in a matrix.

Loc

The `Loc()` function creates a matrix of positions that locate where **A** is true (nonzero and nonmissing).

```
A = [0 1 . 3 4 0];
B = [2 0 0 2 5 6];
```

The following example returns the indices for the values of **A** that are nonmissing and nonzero.

```
I = Loc( A );
    [2, 4, 5]
```

The following example returns the indices for the values of **A** that are less than the corresponding values of **B**. Note that the two matrices must have the same number of rows and columns.

```
I = Loc( A < B );
    [1, 5, 6]
```

The following script replaces all values less than 4 in **A** with 0.

```
A = [0 1 0 3 4 0];
A[Loc( A < 4 )] = 0;
Show( A );
    A = [0 0 0 0 4 0];
```

The following script returns an empty matrix:

```
Loc( [2 3 4 5 6] > 7 );
    [](0, 1)
```

Loc Nonmissing

The `Loc Nonmissing()` function returns a vector of row numbers in a matrix that do not contain any missing values. For example,

```
A = [1 2 3, 4 . 6, 7 8 ., 8 7 6];
Loc Nonmissing( A );
    [1, 4]
```

Loc Min and Loc Max

The `Loc Min()` and `Loc Max()` functions return the position of the first occurrence of the minimum and maximum elements of a matrix. Elements of a matrix are numbered consecutively, starting in the first row, first column, and proceeding left to right.

```
A = [1 2 2, 2 4 4, 1 1 1];
B = [6, 12, 9];
Show( Loc Max( A ) );
Show( Loc Min( B ) );
   Loc Max(A) = 5;
   Loc Min(B) = 1;
```

Loc Sorted

The `Loc Sorted()` function is mainly used to identify the interval that a number lies within. The function returns the position of the highest value in **A** that is less than or equal to the value in **B**. The resulting vector contains an item for each element in **B**.

```
A = [10 20 30 40];
B = [35];
Loc Sorted( A, B );
   [3]
A = [10 20 40];
B = [35 5 45 20];
Loc Sorted( A, B );
   [2, 1, 3, 2]
```

Here's an example that defines a list of strings. `Loc Sorted()` identifies the string at the seventh interval and puts it in a list.

```
{"No Fat", "Some Fat", "Low Fat", "Normal Fat", "Medium Fat", "High Fat"}
[Loc Sorted( [0, 1, 5, 10, 20, 25], 7)]
   {"Low Fat"}
```

To convert the returned value to a string, include the subscript [1] at the end of the expression:

```
{"No Fat", "Some Fat", "Low Fat", "Normal Fat", "Medium Fat", "High Fat"}
[Loc Sorted( [0, 1, 5, 10, 20, 25], 7)][1]
   "Low Fat"
```

Notes:

- **A** must be sorted in ascending order.

- The returned values are always 1 or greater. If the element in **B** is smaller than all of the elements in **A**, then the function returns a value of 1. If the element in **B** is greater than all of the elements in **A**, then the function returns *n*, where *n* is the number of elements in **A**.

Ranking and Sorting

The Rank() function returns the positions of the numbers in a vector or list, as if the numbers were sorted from lowest to highest.

```
E = [1 -2 3 -4 0 5 1 8 -7];
R = Rank( E );
    [9, 4, 2, 5, 7, 1, 3, 6, 8]
```

If **E** were sorted from lowest to highest, the first number would be -7. The position of -7 in **E** is 9.

The original matrix **E** can then be sorted using the matrix **R** as subscripts to **E**.

```
sortedE = E[R];
    [-7,- 4, -2, 0, 1, 1, 3, 5, 8]
```

The Ranking Tie() function returns ranks for the values in a vector or list, with ranks for ties averaged. Similarly, Ranking() returns ranks for the values in a vector or list, but the ties are ranked arbitrarily.

```
E = [1 -2 3 -4 0 5 1 8 -7];
Ranking Tie( E );
    [5.5, 3, 7, 2, 4, 8, 5.5, 9, 1]

E = [1 -2 3 -4 0 5 1 8 -7];
Ranking( E );
    [5, 3, 7, 2, 4, 8, 6, 9, 1]
```

The Sort Ascending() and Sort Descending() functions sort vectors.

```
E = [1 -2 3 -4 0 5 1 8 -7];
Sort Ascending( E );
    [-7 -4 -2 0 1 1 3 5 8]

E = [1 -2 3 -4 0 5 1 8 -7];
Sort Descending( E );
    [8 5 3 1 1 0 -2 -4 -7]
```

If the argument is not a vector or list, an error message is generated.

Special Matrices

Construct an Identity Matrix

The Identity() function constructs an identity matrix of the dimension that you specify. An identity matrix is a square matrix of zeros except for a diagonal of ones. The only argument specifies the dimension.

```
Identity( 3 );
```

```
[1 0 0,
0 1 0,
0 0 1]
```

Construct a Matrix with Specific Values

The `J()` function constructs a matrix with the number of rows and columns that you specify as the first two arguments, whose elements are all the third argument, for example:

```
J( 3, 4, 5 );
[5 5 5 5,
5 5 5 5,
5 5 5 5]
J( 3, 4, Random Normal() ); // your results will differ
[0.407709113182904 1.67359154091978 1.00412665221308 0.240885679837327,
−0.557848036549455 −0.620833861982722 0.877166783247633 1.50413740148892,
−2.09920574748608 −0.154797501010655 0.0463943433032137 0.064041826393316]
```

Create a Diagonal Matrix

The `Diag()` function creates a diagonal matrix from a square matrix (having an equal number of rows and columns) or a vector. A diagonal matrix is a square matrix whose nondiagonal elements are zero.

```
D = [1 −1 1];
Diag( D );
[1 0 0,
0 −1 0,
0 0 1]
Diag([1, 2, 3, 4]);
[1 0 0 0,
0 2 0 0,
0 0 3 0,
0 0 0 4]

A = [1 2,3 4];
f = [5];
D = Diag( A, f );
[1 2 0
3 4 0
0 0 5]
```

In the third example, at first glance, not all of the nondiagonal elements are zero. Using matrix notation, the matrix can be expressed as follows:

```
[A 0,
0` f]
```

Where A and f are the matrices from the example, and **0** is a column vector of zeros.

Create a Column Vector from Diagonal Elements

The `VecDiag()` function creates a column vector from the diagonal elements of a matrix.

```
v = Vec Diag(
[1 0 0 1, 5 3 7 1, 9 9 8 8, 1 5 4 3]);
  [1, 3, 8, 3]
```

Calculate Diagonal Quadratic Forms

The `Vec Quadratic()` function calculates the hats in regression that go into the standard errors of prediction or the Mahalanobis or T2 statistics for outlier distances. `Vec Quadratic(Sym, X)` is equivalent to calculating `Vec Diag(X*Sym*X`)`. The first argument is a symmetric matrix, usually an inverse covariance matrix. The second argument is a rectangular matrix with the same number of columns as the symmetric matrix argument.

Return the Sum of Diagonal Elements

The `Trace()` function returns the sum of the diagonal elements for a square matrix.

```
D = [0 1 2, 2 1 0, 1 2 0];
Trace( D ); // returns 1
```

Generate a Row Vector of Integers

The `Index()` function generates a row vector of integers from the first argument to the last argument. A double colon `::` is the equivalent infix operator.

```
6::10;
  [6 7 8 9 10]
Index( 1, 5 );
  [1 2 3 4 5]
```

The optional *increment* argument changes the default increment of +1.

```
Index( 0.1, 0.4, 0.1 );
  [0.1, 0.2, 0.3, 0.4]
```

The increment can also be negative.

```
Index( 6, 0, -2 );
  [6, 4, 2, 0]
```

The default value of the increment is 1, or -1 if the first argument is higher than the second.

Reshape a Matrix

The `Shape()` function reshapes an existing matrix across rows to be the specified dimensions. The following example changes the 3x4 matrix `a` into a 12x1 matrix:

```
a = [1 1 1, 2 2 2, 3 3 3, 4 4 4];
Shape( a, 12, 1 );
  [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]
```

Summarize Columns of a Matrix

There are several functions that return a row vector based on summary values for each column in a matrix.

```
mymatrix = [11 22, 33 44, 55 66];
V Max( mymatrix ); // returns the maximum of each column
  [55 66]
V Min( mymatrix ); // returns the minimum of each column
  [11 22]
V Mean( mymatrix ); // returns the mean of each column
  [33 44]
V Median( mymatrix ); // returns the median of each column
  [33 44]
V Quantile( mymatrix, 0.75 ); // returns the 0.75 quantile of each column
  [55 66]
V Sum( mymatrix ); // returns the sum of each column
  [99 132]
V Std( mymatrix ); // returns the standard deviations of each column
  [22 22]
```

Create Design Matrices

The `Design()` function creates a matrix of design columns for a vector or list. There is one column for each unique value in the vector or list. The design columns have the elements 0 and 1. For example, **x** below has values 1, 2, and 3, then the design matrix has a column for 1s, a column for 2s, and a column for 3s. Each row of the matrix has a 1 in the column representing that row's value. So, the first row (1) has a 1 in the 1s column (the first column) and 0s elsewhere; the second row (2) has a 1 in the 2's column and 0s elsewhere; and so on.

```
x = [1, 2, 3, 2, 1];
Design( x );
  [1 0 0,
   0 1 0,
   0 0 1,
   0 1 0,
   1 0 0]
```

A variation is the `Design Nom()` or `Design F()` function, which removes the last column and subtracts it from the others. Therefore, the elements of `Design Nom()` or `Design F()` matrices are 0, 1, and –1. And the `Design Nom()` or `Design F()` matrix has one less column than the vector or list has unique values. This function makes full-rank versions of design matrices for effects.

```
x = [1, 2, 3, 2, 1];
Design Nom( x );
    [1 0, 0 1, -1 -1, 0 1, 1 0]
```

`Design Nom()` is further demonstrated in the "ANOVA Example" on page 235.

To facilitate ordinal factor coding, use the `Design Ord()` function. This function produces a full-rank coding with one less column than the number of unique values in the vector or list. The row for the lowest value in the vector or list is all zeros. Each succeeding value adds an additional 1 to the row of the design matrix.

```
x = [1, 2, 3, 4, 5, 6];
Design Ord( x );
    [0 0 0 0 0,
    1 0 0 0 0,
    1 1 0 0 0,
    1 1 1 0 0,
    1 1 1 1 0,
    1 1 1 1 1]
```

`Design()`, `Design Nom()`, and `Design Ord()` support a second argument that specifies the levels to be looked up and their order. This feature allows design matrices to be created one row at a time.

- `Design( values, levels )` creates a design matrix of indicator columns.
- `DesignNom( values, levels )` creates a full-rank design matrix of indicator columns.

Notes:

- The *values* argument can be a single element or a matrix or list of elements.
- The *levels* argument can be a list or matrix of levels to be looked up.
- The result has the same number of rows as there are elements in the *values* argument.
- The result always has the same number of columns as there are items in the *levels* argument. In the case of `Design Nom()` and `Design Ord()`, there is one less column than the number of items in the *levels* argument.
- If a value is not found, the whole row is zero.

Examples

```
Design( 20, [10 20 30] );
    [0 1 0]
Design( 30, [10 20 30] );
```

```
   [0 0 1]
Design Nom( 20, [10 20 30] );
   [0 1]
Design Nom( 30, [10 20 30] );
   [-1 -1]
Design Ord( 20, [10 20 30] );
   [1 0]
Design( [20, 10, 30, 20], [10 20 30] );
   [0 1 0,
   1 0 0,
   0 0 1,
   0 1 0]
Design({"b", "a", "c", "b"}, {"a", "b", "c"});
   [0 1 0,
   1 0 0,
   0 0 1,
   0 1 0]
```

Find the Direct Product

The `Direct Product()` function finds the direct product (or Kronecker product) of two
square matrices. The direct product of a m x m matrix and a n x n matrix is the mn x mn matrix
whose elements are the products of numbers, one from **A** and one from **B**.

```
G = [1 2, 3 5];
H = [2 3, 5 7];
Direct Product( G, H );
   [2 3 4 6,
   5 7 10 14,
   6 9 10 15,
   15 21 25 35]
```

The `H Direct Product()` function finds the row-by-row direct product of two matrices with
the same number of rows.

```
H Direct Product( G, H );
   [2 3 4 6, 15 21 25 35]
```

`HDirect Product()` is useful for constructing the design matrix columns for interactions.

```
XA = Design Nom( A );
XB = Design Nom( B );
XAB = HDirect Product( XA, XB );
X = J( NRow( A ), 1 ) || XA || XB || XAB;
```

Inverse Matrices and Linear Systems

JMP has the following functions for computing inverse matrices: `Inverse()`, `GInverse()`,
and `Sweep()`. The `Solve()` function is used for solving linear systems of equations.

Inverse or Inv

The `Inverse()` function returns the matrix inverse for the square, nonsingular matrix argument. `Inverse()` can be abbreviated `Inv`. For a matrix **A**, the matrix product of **A** and `Inverse(A)` (often denoted $A(A^{-1})$) returns the identity matrix.

```
A = [5 6, 7 8];
AInv = Inv( A );
A*A Inv;
   [1 0,0 1]

A = [1 2, 3 4];
AInv = Inverse( A );
A*AInv;
   [1 1.110223025e-16,0 1]
```

Note: There can be small discrepancies in the results due to floating point limitations, as illustrated in the second example.

GInverse

The (Moore-Penrose) generalized inverse of a matrix **A** is any matrix **G** that satisfies the following conditions:

AGA=A

GAG=G

(AG)´ = AG

(GA)´ = GA

The `GInverse()` function accepts any matrix, including non-square ones, and uses singular-value decomposition to calculate the Moore-Penrose generalized inverse. The generalized inverse can be useful when inverting a matrix that is not full rank. Consider the following system of equations:

$$x + 2y + 2z = 6$$
$$2x + 4y + 4z = 12$$
$$x + y + z = 1$$

Find the solution to this system using the following script:

```
A = [1 2 2, 2 4 4, 1 1 1];
B = [6, 12, 1];
Show( GInverse( A ) * B );
   GInverse(A) * B = [-4, 2.5, 2.5];
```

Solve

The `Solve()` function solves a system of linear equations. `Solve()` finds the vector **x** so that $x = A^{-1}b$ where *A* equals a square, nonsingular matrix and *b* is a vector. The matrix **A** and the vector **b** must have the same number of rows. `Solve(A,b)` is the same as `Inverse(A)*b`.

```
A = [1 -4 2, 3 3 2, 0 4 -1];
b = [1, 2, 1];
x = Solve( A, b );
  [-16.9999999999999, 4.99999999999998, 18.9999999999999]
A*x;
  [1, 2, 0.999999999999997]
```

Note: There can be small discrepancies in the results due to floating point limitations, as illustrated in the example.

Sweep

The `Sweep()` function inverts parts (or *pivots*) of a square matrix. If you sequence through all of the pivots, you are left with the matrix inverse. Normally the matrix must be positive definite (or negative definite) so that the diagonal pivots never go to zero. `Sweep()` does not check whether the matrix is positive definite. If the matrix is *not* positive definite, then it still works, as long as no zero pivot diagonals are encountered. If zero (or near-zero) diagonal pivots are encountered on a full sweep, then the result is a g2 generalized inverse if the zero pivot row and column are zeroed.

About the Sweep Function

Suppose matrix **E** consists of smaller matrix partitions, **A**, **B**, **C**, and **D**:

$$E = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

The syntax for `Sweep()` appears as follows:

```
Sweep( E, [...] );
```

[...] indicates partition A.

This produces the matrix result equivalent to the following:

$$\begin{bmatrix} A^{-1} & A^{-1}B \\ -CA^{-1} & D - CA^{-1}B \end{bmatrix}$$

Notes:

- The submatrix in the **A** position becomes the inverse.
- The submatrix in the **B** position becomes the solution to Ax = B.
- The submatrix in the **C** position becomes the solution to xA = C.

Use of the Sweep Function

Sweep() is sequential and reversible:

- A = Sweep(A, {i,j}) is the same as A = Sweep(Sweep(A, i), j). It is sequential.
- A = Sweep(Sweep(A, i), i) restores **A** to its original values. It is reversible.

If you have a cross-product matrix partitioned as follows:

$$C = \begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}$$

Then after sweeping through the indices of X'X, the result appears as follows:

$$\begin{bmatrix} (X'X)^{-1} & (X'X)^{-1}X'y \\ -yX(X'X)^{-1} & y'y-y'X(X'X)^{-1}X'y \end{bmatrix}$$

The partitions are recognizable to statisticians as follows:

- the least squares estimates for the model Y = Xb + e in the upper right
- the sum of squared errors in the lower right
- a matrix proportional to the covariance of the estimates in the upper left

The Sweep function is useful in computing the partial solutions needed for stepwise regression.

The *index* argument is a vector that lists the rows (or equivalently the columns) on which you want to sweep the matrix. For example, if **E** is a 4x4 matrix, to sweep on all four rows to get $\mathbf{E}^{-1}$ requires these commands:

```
E = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
Sweep( E, [1, 2, 3, 4] );
   [0.56 -0.44 -0.02 -0.02,
   -0.44 0.56 -0.02 -0.02,
   -0.02 -0.02 0.34 -0.16,
   -0.02 -0.02 -0.16 0.34]
Inverse( E ); // notice that these results are the same
   [0.56 -0.44 -0.02 -0.02,
   -0.44 0.56 -0.02 -0.02,
```

```
    -0.02 -0.02 0.34 -0.16,
    -0.02 -0.02 -0.16 0.34]
```

Note: For a tutorial on Sweep(), and its relation to the Gauss-Jordan method of matrix inversion, see Goodnight, J.H. (1979) "A tutorial on the SWEEP operator." *The American Statistician*, August 1979, Vol. 33, No. 3. pp. 149–58.

Sweep() is further demonstrated in the

Determinant

The Det() function returns the determinant of a square matrix. The determinant of a 2 × 2 matrix is the difference of the diagonal products, as demonstrated below. Determinants for *n* × *n* matrices are defined recursively as a weighted sum of determinants for (*n* - 1) × (*n* - 1) matrices. For a matrix to have an inverse, the determinant must be nonzero.

$$\begin{vmatrix} 1 & 2 \\ 3 & 5 \end{vmatrix} = (1 \cdot 5) - (3 \cdot 2) = -1$$

```
F = [1 2, 3 5];
Det( F );
   -1
```

Decompositions and Normalizations

This section contains functions that calculate eigenvalues and eigenvectors and functions that decompose matrices.

Eigenvalues

The Eigen() function performs eigenvalue decomposition of a symmetric matrix. Eigenvalue decompositions are used in many statistical techniques, most notably in principal components and canonical correlation, where the transformation associated with the largest eigenvalues are transformations that maximize variances.

Eigen() returns a list of matrices. The first matrix in the returned list is a column vector of eigenvalues; the second matrix contains eigenvectors as the columns.

```
A = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
Eigen( A );
  {[10, 5, 2, 1][0.632455532033676 - 0.316227766016838 - 2.77555756156289e-16
     -0.707106781186547, 0.632455532033676 - 0.316227766016837 -
     1.66533453693773e-16 0.707106781186547, 0.316227766016838
     0.632455532033676 0.707106781186548 0, 0.316227766016837
     0.632455532033676 - 0.707106781186547 0]}
```

Since the function returns a list of matrices, you might want to assign it to a list of two global variables. That way, the column vector of eigenvalues is assigned to one variable, and the matrix of eigenvectors is assigned to another variable:

```
{evals, evecs} = Eigen( A );
```

For some $n \times n$ matrix **A**, eigenvalue decomposition finds all λ (lambda) and vectors **x**, so that the equation Ax = λx has a nonzero solution **x**. The λ's are called eigenvalues, and the corresponding **x** vectors are called eigenvectors. This is equivalent to solving (A - λI)**x** = 0. You can reconstruct **A** from eigenvalues and eigenvectors by a statement like the following:

```
newA = evecs * Diag( evals ) * evecs`;
  [5.00000000000001 4 1 1,
  4 5 1 1,
  1 1 4 2,
  1 1 2 4]
```

Note the following about eigenvalues and eigenvectors:

- The eigenvector matrices are orthonormal, such that the inverse is the transpose: E′E = EE′ = I.

- Eigenvectors are uniquely determined only if the eigenvalues are unique.

- Zero eigenvalues correspond to singular matrices.

- Inverses can be obtained by inverting the eigenvalues and reconstituting with the eigenvectors. Moore-Penrose generalized inverses can be formed by inverting the nonzero eigenvalues and reconstituting. (See "GInverse" on page 225.)

Note: You must decide whether a very small eigenvalue is effectively zero.

- The eigenvalue decomposition enables you to view any square-matrix multiplication as the following:
  - a rotation (multiplication by an orthonormal matrix)
  - a scaling (multiplication by a diagonal matrix)
  - a reverse rotation (multiplication by the orthonormal inverse, which is the transpose), or in notation:

```
A * x = E` * Diag( M ) * E * x;
```

E rotates, `Diag( M )` scales, and `E`` reverse-rotates.

Cholesky Decomposition

The `Cholesky()` function performs Cholesky decomposition. A positive semi-definite matrix **A** is re-expressed as the product of a nonsingular, lower-triangular matrix **L** and its transpose: **L*L' = A**.

```
E = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
```

```
L = Cholesky( E );
    [2.23606797749979 0 0 0,
    1.788854381999832 1.341640786499874 0 0,
    0.447213595499958 0.149071198499986 1.9436506316151 0,
    0.447213595499958 0.149071198499986 0.914659120760047 1.71498585142509]
```

To verify the results, enter the following:

```
L*L`;
    [5 4 1 1,
    4 5 1 1,
    1 1 4 2,
    1 1 2 4]
```

About Cholesky Decomposition

`Cholesky()` is useful for reconstituting expressions into a more manageable form. For example, eigenvalues are available only for symmetric matrices in JMP, so the eigenvalues of the product **AB** could not be obtained directly. (Although **A** and **B** can be symmetric, the product is not symmetric.) However, you can usually rephrase the problem in terms of eigenvalues of **L′BL** where **L** is the Cholesky root of **A**, which has the same eigenvalues.

Another use of `Cholesky()` is in reordering matrices within `Trace()` expressions. Expressions such as `Trace(A*B*A`)` might involve huge operations counts if **A** has many rows. However, if **B** is small and can be factored into LL′ by Cholesky, then it can be reformulated to `Trace(A*L*L`*A`)`. The resulting matrix is equal to `Trace(L`A`*AL)`. This expression involves a much smaller number of operations, because it consists of only the sum of squares of the **AL** elements.

Use the function `Chol Update()` to update a Cholesky decomposition. If **L** is the Cholesky root of a $n \times n$ matrix **A**, then after using `cholUpdate(L, C, V)`, **L** will be replaced with the Cholesky root of A+V*C*V'. **C** is an $m \times m$ symmetric matrix and **V** is an $n \times m$ matrix.

Examples

Manually update the Cholesky decomposition, as follows:

```
exS = [16 1 0 11 -1 12, 1 11 -1 1 -1 1, 0 -1 12 -1 1 0, 11 1 -1 11 -1 9, -1 -1
    1 -1 9 -1, 12 1 0 9 -1 12];
exAchol = Cholesky( exS ); // conducts the Cholesky decomposition

// add two column vectors to the design matrix
exV = [1 1, 0 0, 0 1, 0 0, 0 0, 0 1];

/* The first column vector is added to one of the rows in the design matrix.
The second column vector is subtracted from one of the rows in the design
    matrix. */
exC = [1 0, 0 -1];
exAnew = exS + exV * exC * exV`;
```

```
    // update the Cholesky decomposition manually
    exAcholnew = Cholesky( exAnew );
```

Instead of manual updating, use Chol Update() to update the Cholesky decomposition, as follows:

```
    // update the Cholesky decomposition more efficiently
    exAcholnew_test = Chol Update( exAchol, exV, exC );
    Show( exAcholnew_test );
        exAcholnew_test =
        [ 4 0 0 0 0 0,
        0.25 3.30718913883074 0 0 0 0, ...]

    // results are the same as the manual process
    Show( exAcholnew );
        exAcholnew=
        [ 4 0 0 0 0 0,
        0.25 3.30718913883074 0 0 0 0, ...]
```

Singular Value Decomposition

The SVD() function finds the singular value decomposition of a matrix. That is, for a matrix **A**, SVD() returns a list of three matrices **U**, **M**, and **V**, so that U*diag(M)*V`=A.

Notes:

- When **A** is taller than it is wide, **M** is more compact, without extra zero diagonals.

- Singular value decomposition re-expresses **A** in the form USV′, where:

  - **U** and **V** are matrices that contain orthogonal column vectors (perpendicular, statistically independent vectors)

  - **S** is a $n \times n$ diagonal matrix containing the nonnegative square roots of the eigenvalues of **A′A**, the singular values of **A**.

- Singular value decomposition is the basis of correspondence analysis.

Example

```
    A = [1 2 1 0, 2 3 0 1, 1 0 1 5, 0 1 5 1];
    {U, M, V} = SVD( A );
    newA = U * Diag( M ) * V`;
        [1 2 0.999999999999997 -2.99456640040496e-15,
        2 3 -1.17505831453979e-15 1,
        0.999999999999997 -2.16851276518826e-15 0.999999999999999 5,
        2.22586706011274e-15 1 5 0.999999999999997]
```

Orthonormalization

The `Ortho()` function orthogonalizes the columns and then divides the vectors by their magnitudes to normalize them. This function uses the Gram-Schmidt method. The column vectors of orthogonal matrices are unit-length and are mutually perpendicular (their dot products are zero).

```
B = Ortho( [1 -1, 1 0, 0 1] );
    [0.408248290463863 -0.707106781186548, 0.408248290463863 0.707106781186548,
       -0.816496580927726 3.14018491736755e-16]
```

To verify that these vectors are orthogonal, multiply **B** by its transpose, which should yield the identity matrix.

```
C = B` * B;
    [1 -3.119061760824e-16, -3.119061760824e-16 1]
```

By default, vectors are normalized, meaning that they are divided by their magnitudes, which scales them to have length 1. Include the option `Scaled(0)` to turn scaling off:

```
Ortho( [1 -1, 1 0, 0 1], Scaled( 0 ) );
    [0.408248290463863 -0.353553390593274, 0.408248290463863 0.353553390593274,
       -0.816496580927726 1.57009245868377e-16]
```

To create vectors whose elements sum to zero, include the `Centered(1)` option. This option is useful when constructing a matrix of contrasts.

```
result = Ortho( [1 -1, 1 0, 0 1], Centered( 1 ) );
    [0.408248290463863 -0.707106781186548, 0.408248290463863 0.707106781186548,
       -0.816496580927726 3.14018491736755e-16]
```

To verify that the elements of each column sum to zero, premultiply by a vector of ones to sum the columns.

```
J(1, 3) * result;
    [1.11022302462516e-16 2.02996189274239e-16]
```

Orthogonal Polynomials

The `Ortho Poly()` function returns orthogonal polynomials for a vector of indices. The polynomial order is specified as a function argument. Orthogonal polynomials can be useful for fitting polynomial models where some regression coefficients might be highly correlated.

```
Ortho Poly( [1 2 3], 2 );
    [-0.707106781186548 0.408248290463862, 0 -0.816496580927726,
       0.707106781186548 0.408248290463864]
```

The polynomial order must be less than the dimension of the vector. Use the `Scaled(1)` option to produce vectors of unit length, as described in "Orthonormalization" on page 232.

QR Decomposition

QR() factorization is useful for numerically stable matrix work. QR() returns a list of two matrices. Here is an example:

```
{Q, R} = QR( [11 22, 33 44] );
```

Q and R hold the results. For a $m \times n$ matrix, QR() creates an orthogonal $m \times m$ matrix Q and an upper triangular $m \times n$ matrix R, so that X = Q*R.

Update Inverse Matrices

To add or drop one or more rows in an inverse of an **M`M** matrix, use the `Inv Update(S, X, w)` function. Updating inverse matrices is helpful in drop-1 influence diagnostics and also in candidate design evaluation.

Notes:

- The first argument, **S**, is the matrix to be updated.
- The second argument, **X**, is the matrix of rows to be added or dropped.
- The third argument, w, is either 1 to add or −1 to delete the row or rows.
- Multiple rows can be added or deleted.

Using the `Inv Update(S, X, w)` function is equivalent to calculating the following:

```
S – w * S * X` * Inv( I + w * X * S * X` ) * X * S
```

Where **I** is an identity matrix and `Inv(A)` is an inverse matrix of **A**.

Build Your Own Matrix Functions

You can store your own operations in macros. See "Macros" on page 265 in the "Programming Methods" chapter. Similarly, you can create custom matrix functions. For example, you can make your own matrix operation called Mag() to find the magnitude of a vector:

```
mag = Function( {x},
   Sqrt( x` * x )
);
```

Similarly, you could create a function called Normalize to divide a vector by its magnitude:

```
normalize = Function( {x},
   x / Sqrt( x` * x )
);
```

Statistical Examples

This section contains statistical examples of using matrices.

Regression Example

Suppose that you want to implement your own regression calculation, rather than use the facilities built into JMP. Because of the compact matrix notation, it might require only a few lines of code:

```
Y = [98, 112.5, 84, 102.5, 102.5, 50.5, 90, 77, 112, 150, 128, 133, 85, 112];
X = [65.3, 69, 56.5, 62.8, 63.5, 51.3, 64.3, 56.3, 66.5, 72, 64.8, 67, 57.5,
    66.5];
X = J( N Row( X ), 1 ) || X; // put in an intercept column of 1s
beta = Inv( X` * X ) * X` * Y; // the least square estimates
resid = Y - X * beta; // the residuals, Y - predicted
sse = resid` * resid; // sum of squared errors
Show( beta, sse );
```

This could be expanded into a script that gets its data from a data table, calculates additional results, and shows the results in a report window:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

// get data into matrices
x = (Column( "Age" ) << Get Values ) || (Column( "Height" ) << Get Values);
x = J(N Row( x ), 1, 1) || x;
y = Column( "Weight" ) << Get Values;

// regression calculations
xpxi = Inv( x` * x );
beta = xpxi * x` * y; // parameter estimates
resid = y - x * beta; // residuals
sse = resid` * resid; // sum of squared errors
dfe = N Row( x ) - N Col( x );  // degrees of freedom
mse = sse / dfe; // mean square error, error variance estimate

// additional calculations on estimates
stdb = Sqrt( Vec Diag( xpxi ) * mse ); // standard errors of estimates
alpha = .05;
qt = Students t Quantile( 1 - alpha / 2, dfe );
betau95 = beta + qt * stdb; // upper 95% confidence limits
betal95 = beta - qt * stdb; // lower 95% confidence limits
tratio = beta :/ stdb; // Student's T ratios
probt = ( 1 - TDistribution( Abs( tratio ), dfe ) ) * 2; // p-values
```

```
// present results
New Window( "Big Class Regression",
  Table Box(
    String Col Box( "Term",{"Intercept", "Age", "Height"} ),
    Number Col Box( "Estimate", beta ),
    Number Col Box( "Std Error", stdb ),
    Number Col Box( "TRatio", tratio ),
    Number Col Box( "Prob>|t|", probt ),
    Number Col Box( "Lower95%", beta195 ),
    Number Col Box( "Upper95%", betau95 ) ) );
```

ANOVA Example

You can implement your own one-way ANOVA. This example presents a problem involving a three-level factor indicating Low, Medium, and High doses and a response measurement. Therefore, this example solves the general linear model, as follows:

$$Y = a + bX + e$$

Where:

- **Y** is a vector of responses
- *a* is the intercept term
- **b** is a vector of coefficients
- **X** is a design matrix for the factor
- *e* is an error term

```
factor = [1, 2, 3, 1, 2, 3, 1, 2, 3];
y = [1, 2, 3, 4, 3, 2, 5, 4, 3];
```

First, build a design matrix for the factor:

```
Design Nom( factor );
  [1 0, 0 1, -1 -1, 1 0, 0 1, -1 -1, 1 0, 0 1, -1 -1]
```

Next, add a column of 1s to the design matrix for the intercept term. You can do this by concatenating J and Design Nom():

```
x = J( 9, 1, 1) || Design Nom( factor );
  [1 1 0,
  1 0 1,
  1 -1 -1,
  1 1 0,
  1 0 1,
  1 -1 -1,
  1 1 0,
  1 0 1,
  1 -1 -1]
```

Now, to solve the normal equation, you need to construct a matrix **M** with partitions:

$$\begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}$$

You can construct matrix **M** in one step by concatenating the pieces:

```
M = ( x` * x || x` * y )
         |/
  ( y` * x || y` * y );
  [9 0 0 27,
    0 6 3 2,
    0 3 6 1,
    27 2 1 93]
```

Now, sweep **M** over all the columns in **X'X** for the full fit model, and over the first column only for the intercept-only model:

```
FullFit = Sweep( M, [1, 2, 3] ); // full fit model
InterceptOnly = Sweep( M, [1] ); // model with intercept only
```

Recall that some of the standard ANOVA results are calculated by comparing the results of these two models. This example focuses on the full fit model, which produces this swept matrix:

```
[0.111111111111111 0 0 3,
0 0.222222222222222 -0.11111111111111 0.333333333333333,
0 -0.11111111111111 0.222222222222222 0,
-3 -0.33333333333333 0 11.33333333333333]
```

Examine the model coefficients from the upper right partition of the matrix. The lower left partition is the same, except that the signs are reversed: 3, 0.333, 0. The results can be interpreted as follows:

- The coefficient for the intercept term is 3.

- The coefficient for the first level of the factor is 0.333.

- The coefficient for the second level is 0.

- Because of the use of Design Nom(), the coefficient for the third level is the difference, –0.333.

- The lower right partition of the matrix holds the sum of squares, 11.333.

You could modify this into a generalized ANOVA script by replacing some of the explicit values in the script with arguments. These results match those from the Fit Model platform.

Figure 7.1 ANOVA Report Within Fit Model
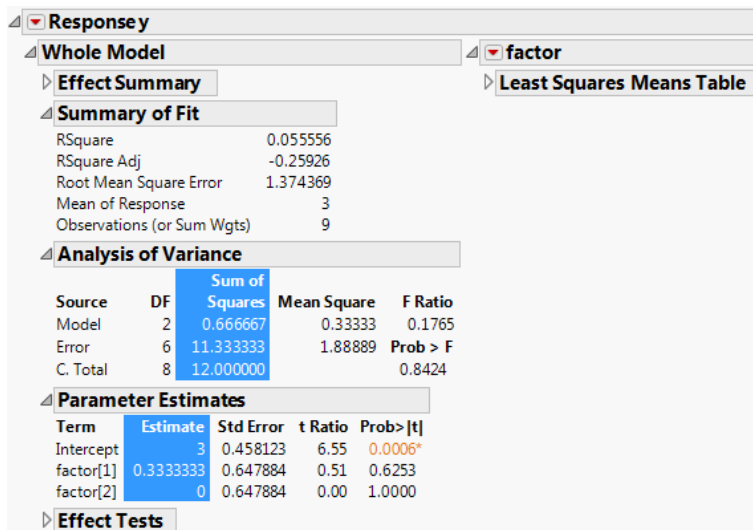
Here is how to construct the report in Figure 7.1:

1. Build a data table (described in the "Data Tables" chapter on page 321):

```
dt = New Table( "Data" );
dt << New Column( "y", Set Values( [1, 2, 3, 4, 3, 2, 5, 4, 3] ) );
dt << New Column( "factor", "Nominal", Values( [1, 2, 3, 1, 2, 3, 1, 2, 3] )
   );
```

2. Run a model (described in "Launch the Analysis and Save the Script to a Window" on page 472 in the "Scripting Platforms" chapter):

```
obj = Fit Model(
   Y( :y ),
   Effects( :factor ),
   Personality( "Standard Least Squares" ),
   Run(
      y << {Plot Actual by Predicted( 0 ), Plot Residual by Predicted( 0 ),
      Plot Effect Leverage( 0 )}
   )
);
```

3. Use JSL techniques for navigating displays (described in "Display Box Object References" on page 524 in the "Display Trees" chapter):

```
ranova = obj << Report;
ranova[Outline Box( 6 )] << Close( 0 );
ranova[Outline Box( 7 )] << Close( 1 );
ranova[Outline Box( 9 )] << Close( 1 );
ranova[Outline Box( 5 ), Number Col Box( 2 )] << Select;
ranova[Outline Box( 6 ), Number Col Box( 1 )] << Select;
```

The result is shown in Figure 7.1.

Associative Arrays

An associative array maps unique keys to values that can be non-unique. An associative array is also called a dictionary, a map, a hash map, or a hash table.

Note the following about keys:

- Keys are placed in quotes.
- The value associated with a key can be a number, date, matrix, list, and so on.
- A key cannot be a floating point number. Only integers are supported.
- You can use matrices as both keys and values, or lists as both keys and values, but you cannot mix the two. In other words, you cannot use a matrix as a key and a list as a value, or the other way around.
- Though associative arrays are not usually ordered, in JMP, keys are returned in alphanumeric order for the purpose of iteration and serialization.

For very large lists, using an associative array is more efficient and faster.

Create Associative Arrays

To create an empty associative array, use the Associative Array() function or [=>].

```
cary = Associative Array();
cary = [=> ];
    Associative array( 0 elements ) assigned.
```

Keys and values can be any JSL objects. Items can be added and changed with subscripting:

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
```

Default Values

A default value determines the value of a key that does not exist in an associative array. If you try to access a key that does not exist in an associative array, an error results. If you define a default value for your associative array, accessing a key that does not exist results in the following:

- adds the key to the associative array
- assigns the default value to the new key
- returns the new key's (default) value instead of an error

If you construct an associative array from a list of strings without assigning values to the keys, then the keys are assigned values of 1. The default value for the associative array is set to 0.

To set the default value:

```
cary = Associative Array();
cary << Set Default Value( "Cary, NC" );
```

To determine whether there is a default value set for an associative array, use the <<Get Default Value message.

```
cary << Get Default Value;
    "Cary, NC"
```

If there is no default value, Empty() is returned.

Besides the Set Default Value message, a default value can be set in the literal constructor using =>value without a key.

```
counts = ["a" => 10,
"b" => 3,
=> 0]; // default value of 0
counts["c"] += 1;
Show( counts );
    counts = ["a" => 10, "b" => 3, "c" => 1, => 0];
```

In the first line, the default value is set to 0. In the second line, the key "c" does not exist in counts. In the output, the key "c" is created with the default value of 0 and then incremented by 1.

Note: If a key's value is the default value, then the key is dropped because any key will return the default value.

Associative Array Constructors

Create an empty associative array:

```
map = [=>];
map = Associative Array();
```

Create an empty associative array with a default value:

```
map = [=>0];
map = Associative Array( 0 );
```

Create an associative array with specific values:

```
map = ["yes" => 0, "no" => 1];
```

Create an associative array with specific values with a default value:

```
map = ["yes" => 0, "no" => 1, => 2];
```

Create an associative array with specific values that contains an associative array:

```
map = ["yes" => 0, "no" => 1, "alt" => ["a" => "1", "b" => "2"]];
```

Create an associative array from a list that contains two lists of a key-value pair:

```
map = Associative Array( {{"yes", 0}, {"no", 1}} );
```

Create an associative array from a list that contains two lists of a key-value pair with a default value:

```
map = Associative Array( {{"yes", 0}, {"no", 1}}, 2 );
```

Create an associative array from a list of keys and a list of values:

```
map = Associative Array( {"yes", "no"}, {0, 1} );
```

Create an associative array from a list of keys and a list of values with a default value:

```
map = Associative Array( {"yes", "no"}, {0, 1}, 2 );
```

Create an associative array from two column references. The first column contains the keys and the second contains the values.

```
map = Associative Array( :name, :height );
```

Create an associative array from two column references with a default value:

```
map = Associative Array(:name, :height, .);
```

Create an associative array from a single list of keys or a single column reference of keys with a default value of 0:

```
set = Associative Array( {"yes", "no"} );
set = Associative Array( :name );
```

Work with Associative Arrays

Find the Number of Keys

To determine the number of keys that an associative array contains, use the `N Items()` function.

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
N Items( cary );
    4
```

Add and Delete Keys and Values

To add or delete key-value pairs from an associative array, use the following functions:

· `Insert()`
· `Insert Into()`
· `Remove()`
· `Remove From()`

Notes:

- `Insert()` and `Remove()` return a named copy of the given associative array with the key-value pair added or removed.

- `Insert Into()` and `Remove From()` add or remove the key-value pairs directly from the given associative array.

- `Insert()` and `Insert Into()` take three arguments: the associative array, a key, and a value.

- `Remove()` and `Remove From()` take two arguments: the associative array and a key.

- If you insert a key with no value provided, the key is assigned a value of 1.

Examples

The following examples illustrate `Insert()` and `Insert Into()`:

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
```

```
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
newcary = Insert( cary, "time zone", "Eastern" );
Show( cary, newcary );
    cary = ["high schools" => {"Cary", "Green Hope", "Panther Creek"},
        "population" => 116244, "state" => "NC", "weather" => "sunny"];
    newcary = ["high schools" => {"Cary", "Green Hope", "Panther Creek"},
        "population" => 116244, "state" => "NC", "time zone" => "Eastern",
        "weather" => "sunny"];

cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
Insert Into(cary, "county", "Wake");
Show( cary );
    cary = ["county" => "Wake", "high schools" => {"Cary", "Green Hope", "Panther
        Creek"}, "population" => 116244, "state" => "NC", "weather" => "sunny"];
```

Note that cary <<Insert is a message sent to an associative array that does the same thing as the function Insert Into(). For example, these two statements are equivalent:

```
cary << Insert( "county", "Wake" );
Insert Into( cary, "county", "Wake" );
```

The following examples illustrate Remove() and Remove From():

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
newcary = Remove( cary, "high schools" );
Show( cary, newcary );
    cary = ["high schools" => {"Cary", "Green Hope", "Panther Creek"},
        "population" => 116244, "state" => "NC", "weather" => "sunny"];
    newcary = ["population" => 116244, "state" => "NC", "weather" => "sunny"];
Remove From( cary, "weather" );
Show( cary );
    cary = ["high schools" => {"Cary", "Green Hope", "Panther Creek"},
        "population" => 116244, "state" => "NC", "weather" => "sunny"];
    newcary = ["population" => 116244, "state" => "NC", "weather" => "sunny"];
    cary = ["high schools" => {"Cary", "Green Hope", "Panther Creek"},
        "population" => 116244, "state" => "NC"];
```

Note that aa << Remove is a message sent to an associative array that does the same thing as the function Remove From(). For example, these two statements are equivalent:

```
cary << Remove( "weather" );
Remove From( cary, "weather" );
```

Find Keys or Values in an Associative Array

To determine whether a certain key is contained within an associative array, use the Contains() function.

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
Contains(cary, "high schools");
    1
Contains(cary, "lakes");
    0
```

To obtain a list of all keys contained in an associative array, use the << Get Keys message.

```
cary << Get Keys;
    {"high schools", "population", "state", "weather"}
```

To obtain a list of all values contained in an associative array, use the <<Get Values message.

```
cary << Get Values;
    {{"Cary", "Green Hope", "Panther Creek"}, 116244, "NC", "sunny"}
```

If you want to see only the values for certain keys, you can specify them as arguments. The keys must be given as a list.

```
cary << Get Values({"state", "population"});
    {"NC", 116244}
```

To see a value for a single key, use the <<Get Value message. Specify only one key and do not place it in a list.

```
cary << Get Value("weather");
    "sunny"
```

To obtain a list of all key-value pairs in an associative array, use the <<Get Contents message.

```
cary << Get Contents;
    {{"high schools", {"Cary", "Green Hope", "Panther Creek"}},
    {"population", 116244},
    {"state", "NC"},
    {"weather", "sunny"}}
```

Note: Using the <<Get Contents message, the returned list does not include the default value. Keys are listed alphabetically.

Iterate through an Associative Array

To iterate through as associative array, use the <<First and <<Next messages. <<First returns the first key in the associative array. <<Next(key) returns the key that follows the *key* that is given as the argument.

The following example removes all key-value pairs from the associative array cary, leaving an empty associative array:

```
currentkey = cary << First;
total = N Items( cary );
For( i = 1, i <= total, i++,
   nextkey = cary << Next( currentkey );
   Remove From( cary, currentkey );
   currentkey = nextkey;
);
Show( cary );
   cary = [=>];
```

Applications for Associative Arrays

You can use associative arrays to quickly and efficiently perform other tasks.

Get the Unique Values from a Data Table Column

Note: The strategy described here is not useful for columns with floating point numbers. Use Summarize instead. See "Store Summary Statistics in Global Variables" on page 364 in the "Data Tables" chapter.

A key can exist only once in an associative array, so putting a column's values into one automatically results in the unique values. For example, the Big Class.jmp sample data table contains 40 rows. To see how many unique values are in the column height, run this script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
unique heights = Associative Array( dt:height );
nitems( unique heights );
   17
```

There are only 17 unique values for height. You can use those unique values by getting the keys:

```
unique heights << Get Keys;
   {51, 52, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70}
```

Note: This is possible because you can use any JMP data type as keys in an associative array, not only strings.

Using an associative array to discover unique values in a column is efficient and fast. The following script takes some time to create a data table with 100,000 rows. Finding the 39 unique values takes very little time.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
nms = dt:name << Get Values;
dtbig = New Table( "Really Big Class",
  New Column( "name",
    Character,
    Set Values( nms[J( 100000, 1, Random Integer( N Items( nms ) ) )] )
  )
);
Wait( 0 );
t1 = Tick Seconds();
Write(
  "\!N# names from Really Big Class = ",
  N Items( Associative Array( dtbig:name ) ),
  ", elapsed time=",
  Tick Seconds() - t1
);
  # names from Really Big Class = 39, elapsed time=0.116666666639503
```

Sort a Column's Values in Lexicographic Order

Because keys are ordered lexicographically, putting the values into an associative array also sorts them. For example, the <<Get Keys message returns the keys (unique values of the names column) in ascending order:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
unique names = Associative Array( dt:name );
unique names << Get Keys;
  {"ALFRED", "ALICE", "AMY", "BARBARA", "CAROL", "CHRIS", "CLAY", "DANNY",
    "DAVID", "EDWARD", "ELIZABETH", "FREDERICK", "HENRY", "JACLYN", "JAMES",
    "JANE", "JEFFREY", "JOE", "JOHN", "JUDY", "KATIE", "KIRK", "LAWRENCE",
    "LESLIE", "LEWIS", "LILLIE", "LINDA", "LOUISE", "MARION", "MARK",
    "MARTHA", "MARY", "MICHAEL", "PATTY", "PHILLIP", "ROBERT", "SUSAN",
    "TIM", "WILLIAM"}
```

Compare Columns in Two Different Data Tables

Using associative arrays, determining which values in one column are not in another column (or determining which values are in both columns) is fast. For example, given two data tables with information about countries, which countries are in both data tables?

Place the columns of each data table that contain country names into associative arrays:

```
dt1 = Open( "$SAMPLE_DATA/BirthDeathYear.jmp" );
dt2 = Open( "$SAMPLE_DATA/World Demographics.jmp" );
aa1 = Associative Array( dt1:Country );
aa2 = Associative Array( dt2:Territory );
```

Use N Items() to see how many countries appear in each data table:

```
N Items(aa1);
   23
N Items(aa2);
   239
```

Use the <<Intersect message to find the common values:

```
aa1 = Associative Array( dt1:Country );
aa1 << Intersect( aa2 );
```

Look at the results:

```
Show(N Items(aa1), aa1 << Get Keys);
   N Items(aa1) = 21;
   aa1 << get keys = {"Australia", "Austria", "Belgium", "France", "Greece",
      "Ireland", "Israel", "Italy", "Japan", "Mauritius", "Netherlands", "New
      Zealand", "Norway", "Panama", "Poland", "Portugal", "Romania",
      "Switzerland", "Tunisia", "United Kingdom", "United States"};
```

This example uses a set operation called intersection. For more examples of using set operations with associative arrays to compare values, see "Associative Arrays in Set Operations" on page 249.

Associative Arrays in Graph Theory

You can use associative arrays for graph theory data structures, such as the following directed graph example:

```
g = Associative Array();
g[1] = Associative Array({1, 2, 4});
g[2] = Associative Array({1, 3});
g[3] = Associative Array({4, 5});
g[4] = Associative Array({4, 5});
g[5] = Associative Array({1, 2});
```

This is a two-level associative array. The associative array g contains five associative arrays (1, 2, 3, 4, and 5). In the containing array g, both the keys (1-5) and the values (the arrays that define the map) are important. In the inner associative arrays, the values do not matter. Only the keys are important.

The associative array represents the graph shown in Figure 7.2:

- node 1 is incident on nodes 1, 2, and 4
- node 2 is incident on nodes 1 and 3

- node 3 is incident on nodes 4 and 5
- node 4 is incident on nodes 4 and 5
- node 5 is incident on nodes 1 and 2

Figure 7.2 Example of a Directed Graph
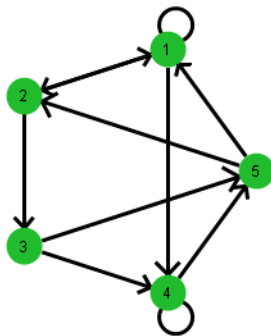
The following depth-first search function can be used to traverse this graph, or any other directed graph represented as an associative array:

```
dfs = Function( {ref, node, visited},
    {chnode, tmp},
    Write( "\!NNode: ", node, ", ", ref[node] << Get Keys );
    visited[node] = 1;
    tmp = ref[node];
    chnode = tmp << first;
    While( !Is Missing( chnode ),
        If( !visited[chnode],
            visited = Recurse( ref, chnode, visited )
        );
        chnode = tmp << Next( chnode );
    );
    visited;
);
```

Notes:

- The first argument is the associative array that contains the map structure.
- The second argument is the node that you want to use as the starting point.
- The third argument is a vector that the function uses to keep track of nodes visited.

To see how the preceding example works, add the following expression to the end of the script:

```
dfs( g, 2, J( N Items( g << Get Keys ), 1, 0 ) );
```

Here is the output:

```
Node 2: {1, 3}
Node 1: {1, 2, 4}
Node 4: {4, 5}
Node 5: {1, 2}
Node 3: {4, 5}
[1, 1, 1, 1, 1]
```

The first five output lines show that starting from node 2, you can reach all the other nodes in the order in which they are listed. Each node also lists the nodes it is incident on (the keys). The value for each key is 1. The final line is a matrix that shows that you can reach each node from 2. If there were nodes that could not be reached from node 2, their values in the matrix would be 0.

Here is how to read the traversal of the nodes:

1. Start at node 2 and go to node 1.

2. From node 1, go to node 4.

3. From node 4, go to node 5.

4. From node 5, go back to node 2, and then to node 3.

Map Script

Here is the script that produced the map shown in Figure 7.2.

```
New Window( "Directed Graph",
    Graph Box(
        Frame Size( 300, 300 ),
        X Scale( -1.5, 1.5 ),
        Y Scale( -1.5, 1.5 ),
        Local( {n = N Items( g ), k = 2 * Pi() / n, r, i, pt, from, to,
            edge, v, d},
            Fill Color( "green" );
            Pen Size( 3 );
            r = 1 / (n + 2);
            For( i = 1, i <= n, i++,
                pt = Eval List( {Cos( k * i ), Sin( k * i )} );
                edges = g[i];
                For( edge = edges << First, !Is Empty( edge ),
                    edge = edges << Next( edge ),
                    to = Eval List( {Cos( k * edge ), Sin( k * edge )} );
                    If( i == edge,
                        Circle( Eval List( 1.2 * pt ), 0.9 * r ), // else
                        v = pt - to;
                        d = Sqrt( Sum( v * v ) );
                        {from, to} = Eval List(
                            {pt * (d - r) / d + to * r / d, pt * r / d + to *
```

```
                        (d - r) / d}
                );
                Arrow( from, to );
            );
        );
        Circle( pt, r, "fill" );
        Text( Center Justified, pt - {0, 0.05}, Char( i ) );
      );
    )
  )
);
```

Associative Arrays in Set Operations

You can also use associative arrays to perform set operations. The following examples show how to take a union of two sets, a difference of two sets, and an intersection of two sets.

First, create three sets and view them in the log:

```
set_y = Associative Array( {"chair", "person", "relay", "snake", "tripod"} );
set_z = Associative Array( {"person", "snake"} );
set_w = Associative Array( {"apple", "orange"} );

// write the sets to the log
Write(
  "\!NExample:\!N\!tset_y = ",
  set_y << Get Keys,
  "\!N\!tset_z = ",
  set_z << Get Keys,
  "\!N\!tset_w = ",
  set_w << Get Keys
);
    Example:
      set_y = {"chair", "person", "relay", "snake", "tripod"}
      set_z = {"person", "snake"}
      set_w = {"apple", "orange"}
```

Union Operation

To find the union of two sets, insert one set into the other:

```
set_z << Insert( set_w );
Write( "\!N\!NUnion operation (set_w, set_z):,\!N\!tset_z = ", set_z << Get
  Keys );
    Union operation (set_w, set_z):,
      set_z = {"apple", "orange", "person", "snake"}
```

Difference Operation

To find the difference of two sets, remove one set from the other:

```
set_y << Remove( set_z );
Write( "\!N\!NDifference operation (set_z from set_y):\!N\!tset_y = ", set_y
    << Get Keys );
    Difference operation (set_z from set_y):
        set_y = {"chair", "relay", "tripod"}
```

Intersect Operation

To find the intersection of two sets, use the aa << Intersect message.

```
set_w << Intersect( set_z );
Write( "\!N\!NIntersect operation (set_w, set_z):\!N\!tset_w = ", set_w << Get
    Keys );
    Intersect operation (set_w, set_z):
        set_w = {"apple", "orange"}
```

Example of Using Set Operations

Given a list of names, which of them are not contained in Big Class.jmp? You can find the answer by taking the difference of the two sets of names.

1. Get the list of names and open the data table:

```
names list = {"ROBERT", "JEFF", "CHRIS", "HARRY"};
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

2. Put the list of names into an associative array:

```
names = Associative Array( names list );
```

3. Perform a difference operation by removing the column values from your list:

```
names << Remove( Associative Array( dt:name ) );
```

4. Look at the result:

```
Write( "\!NWhich of {ROBERT, JEFF, CHRIS, HARRY}, is not in Big Class = ",
    names << Get Keys );
    Which of {ROBERT, JEFF, CHRIS, HARRY}, is not in Big Class = {"HARRY", "JEFF"}
```

Programming Methods

Complex Scripting Techniques and Additional Functions

This chapter includes advanced techniques, such as throwing and catching exceptions, encrypting scripts, and using complex expressions.

Contents

Lists and Expressions

Expressions

This section delves into JSL expressions and introduces several JSL functions that can be used to manipulate JSL expressions. These functions provide JSL programmers with a rich, powerful, meta programming toolkit. One of the benefits of such a toolkit is that it provides JSL programmers with a powerful macro system capability that is similar in functionality to macro systems available in other modern programming languages.

In order to understand JSL expressions, it is useful to keep in mind that everything that happens in JSL is a result of a function call. Function calls come in three varieties:

prefix   The function name comes before its arguments (for example, Foo(m, n)). Most function calls in JSL are of this form.

infix   The function name is between its arguments (for example, m + n). This form is common for mathematical operators.

special   Functions like [ ] (the subscript operator) fall into this category. Since there are only a handful of special functions, they are not discussed further in this section.

Although function calls come in three forms, all function calls can be written in prefix form. It turns out that JMP actually uses the prefix form internally and so, in general, it is very useful to think of JSL expressions in this way.

To illustrate this idea, begin by examining the syntactic structure of JSL expressions a bit more closely. All JSL expressions must conform to a very simple syntactic structure. Each expression consists of a *head* expression and a sequence of zero or more argument expressions. Consider the following JSL expressions, written first in infix form and then in prefix form, to see how this syntactic structure applies:

```
x = 100.1
m & n
1 <= z <= 20
```

Using the prefix form, these expressions are written equivalently:

```
Assign( x, 100.1 )
And( m, n )
Less or Equal( 1, z, 20 )
```

The head of the first example is `Assign` and its arguments are `x` and 100.1. For the second example, the head is `And` and its arguments are `m` and `n`, whereas for the third example, the head is `Less or Equal` and its arguments are 1, `z`, and 20. For these examples, the "=", "&", and "<=" operators are just human friendly ways of representing `Assign`, `And`, and `Less or Equal`.

Examine a few additional simple examples.

```
Assign( 100.1, x )
And( )
Less or Equal( z )
```

These examples also conform to the required syntactic structure, but they differ from the previous examples in that they all produce an error message when they are evaluated. This point leads to an important distinction that you should make when thinking about JSL expressions. That is, you should separate the idea of the syntactic structure of a JSL expression from what happens when the expression is evaluated.

Consider the following examples that present both the human friendly form and the prefix form in each case:

```
x = 100.1 + Log( z )                 // human friendly form
Assign( x, Add( 100.1, Log( z ) ) )  // prefix form
```

This example illustrates that the arguments of a head expression can also be expressions.

```
// human friendly form
x = 0.8;
x ||= Random Uniform( 0.2, 0.6 );

// prefix form
Glue(
    Assign( x, 0.8 ),
    Concat To( x, Random Uniform( 0.2, 0.6 ) )
)
```

This example would typically be referred to as a JSL script, but it is just another case of a JSL expression. Here, the *head* expression is `Glue`, which has two arguments: the expressions `Assign( x, 0.8 )` and `Concat To( x, Random Uniform( 0.2, 0.6 ) )`. This example illustrates two important points:

1. The ";" symbol is another instance of a human friendly operator. In this case, it is for the `Glue` function.

2. There is no distinction between what are typically referred to as a JSL script and a JSL expression.

The remainder of this section covers how to assign an expression to a variable and then how to manipulate, or execute, that expression when needed. For information about how JMP resolves names when evaluating JSL expressions, see the section "Rules for Name Resolution" on page 111 in the "JSL Building Blocks" chapter.

Quoting and Unquoting Expressions

The operators to control when expressions are evaluated are `Expr` and `Eval`, which you should think of as the procrastination and eager operators. `Expr` just copies its argument as an expression, rather than evaluating it. `Eval` does the opposite: it evaluates its argument, and then takes that result and evaluates it again.

`Expr` and `Eval` can be thought of as quoting and unquoting operators, telling JMP when you mean the expression itself, and when you mean the result of evaluating the expression.

The following examples all assume these two assignments:

```
x = 1; y = 20;
```

If you assign the expression x+y to *a*, quoting it as an expression with `Expr`, then whenever *a* is evaluated, it evaluates the expression using the current values of *x* and *y* and returns the result. (Exceptions are the utilities `Show`, `Write`, and `Print`, which do not evaluate expressions for pure name arguments.)

```
x = 1; y = 20;
a = Expr( x + y );
a;
    21
```

If you want the expression that is stored in the name, rather than the result of evaluating the expression, use the `NameExpr` function. See "Retrieve a Stored Expression Rather than its Result" on page 257.

```
x = 1; y = 20;
Show( Name Expr( a ) );
    NameExpr(a) = x + y
```

If you assign an extra level of expression-quoting, then when *a* is evaluated, it unpacks one layer and the result is the expression x+y.

```
x = 1; y = 20;
a = Expr( Expr( x + y ) );
Show( a );
    a = Expr(x + y)
```

If you want the value of the expression, then use `Eval` to unpack all layers:

```
x = 1; y = 20;
Show( Eval( a ) );
    Eval(a) = 21
```

You can do this to any level, for example:

```
x = 1; y = 20;
a = Expr( Expr( Expr( Expr( x + y ) ) ) );
b = a;
   Expr( Expr( x + y ) )
c = Eval( a );
   Expr( x + y )
d = Eval( Eval( a ) );
   x+y
e = Eval( Eval( Eval( a ) ) );
   21
```

For more information about expression handling, see Morgan (2010).

Quote an expression as a string

The JSL Quote() function returns the contents of an expression as a quoted string. Comments and white space in the string are preserved.

The following expression is an example:

```
x = JSL Quote( /* Begin quote. */
For (i = 1, i <= 5, i++,
// Print the value of i.
Print( i );
);
// End expression.
);
Show( x );
```

In the output, the contents of the JSL Quote() function are enclosed in quotes.

```
x = " /* Begin quote. */
For (i = 1, i <= 5, i++,
   // Print the value of i.
Print(i);
);
   // End expression.
";
```

Quoting Expressions

A common use of the Expr() function is to store an expression in a JSL variable. This could be considered as a macro. Consider the following script:

```
dist = Expr( Distribution( Column( height ) ) );
```

To execute the above expression, just use the following:

```
dist;
```

You can use expressions in a loop to execute it:

```
For( i = 0, i < 3, i = i + 1, dist );
```

This loop results in three Distribution platform reports for height.

Use `Eval()` to evaluate an expression explicitly:

```
Eval( dist );
```

Note, however, that in column formulas, `Eval()` only works if it is outermost in the formula. So, for example,

```
Formula( Log( Eval( Column Name( i ) ) ) );
```

would generate an error. Instead, use:

```
Formula( Eval( Substitute( Expr( Log( xxx ) ), Expr( xxx ), Column Name( i ) )
   ) );
```

As another example,

```
Formula( Eval( Column Name( i ) ) + 10 );
```

generates an error, since `Eval()` is actually under the Add function. Instead, use:

```
Formula(Eval(Substitute(Expr(xxx+10), Expr(xxx), column name(i))))
```

Retrieve a Stored Expression Rather than its Result

What if you wanted the symbolic value of a variable (such as the expression `Distribution(Column(height))` stored in `dist` above), rather than the evaluation of it (the actual launched platform)? The `Name Expr` function does this. `Name Expr` retrieves its argument as an expression without evaluating it.

`Expr` returns its argument exactly, whereas `Name Expr` looks up the expression stored in its argument. `Name Expr` "unpacks" just one layer to get the expression, but does not keep unpacking to get the result.

For example, you would need to use this if you had an expression stored in a name and you wanted to edit the expression:

```
popVar = Expr( Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row()
   ) );
   Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row() )

unbiasedPopVar = Substitute( Name Expr( popVar ), Expr( Wild()/N Row() ),
   Expr( (y[i] - Col Mean( y )) ^ 2 / ( N Row() - 1 ) ) );
   Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / (N Row() - 1) )
```

Compare x, Expr(x), NameExpr(x), and Eval(x) after submitting this script:

```
a = 1; b = 2; c = 3;
x = Expr( a + b + c );
```

Table 8.1 Compare Eval, Name Expr, and Expr

Command and result	Explanation
x; *6*	Evaluates *x* to the expression a+b+c, and then evaluates the expression, returning the result, 6 (unpacks *all* layers).
Eval(x); *6*	Equivalent to simply calling *x*. Evaluates *x* to the expression a+b+c, and then evaluates the expression, returning the result, 6 (unpacks *all* layers).
NameExpr(x); *a+b+c*	Returns the expression that was stored in *x*, which is a+b+c (unpacks the *outside* layer).
Expr(x); *x*	Returns the expression x (packs *one* layer).

JSL also supports functions to access and traverse expressions, all of them either a name or a literal expression as an argument. In the following, *expressionArg* is either a single name or a compound expression to be taken literally.

NArg(*expressionArg*) finds the number of arguments in *expressionArg*.

The *expressionArg* can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr().

NArg(*name*) obtains the expression held in name (it is not evaluated) and returns the number of arguments

NArg(*expression*) evaluates expression and returns the number of arguments

NArg(Expr(*expression*)) returns the number of arguments to literal expression.

For example, if aExpr = {a+b,c,d,e+f+g};

- NArg(aExpr) results in 4.
- NArg(Arg(aExpr,4)) results in 3.
- NArg(Expr({1,2,3,4})) results in 4.

Head(*expressionArg*) returns the head of the expression without any arguments. If the expression is an infix, prefix, or postfix special character operator, then it is returned as the functional equivalent.

The *expressionArg* can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr().

For example, if `aExpr = expr(a+b)`;

- `r = Head(aExpr)` results in `Add()`.
- `r = Head (Expr(sqrt(r)))` results in `Sqrt()`.
- `r = Head({1,2,3})` results in `{}` .

`Arg(`*expressionArg*`,`*indexArg*`)` extracts the specified argument of the symbolic expression, resulting in an expression.

For example, `Arg(`*expressionArg*`,i)` extracts the ith argument of *expressionArg*.

The *expressionArg* can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by `Expr()`.

- `Arg(`*name*`,i)` obtains the expression held in *name* (it is not evaluated) and finds the ith argument.
- `Arg(`*expression*`,i)` evaluates *expression* and finds the ith argument.
- `Arg(Expr(`*expression*`),i)` finds the ith argument of *expression*.

As another example, if `aExpr = Expr(12+13*sqrt(14-15))`;

- `Arg(aExpr,1)` yields `12`.
- `Arg(aExpr,2)` yields `13*sqrt(14-15)`.
- `Arg(Expr(12+13*sqrt(14-15)),2)` yields `13*sqrt(14-15)`.

To extract an argument of an argument inside an expression, you can nest Arg commands:

- `Arg(Arg(aExpr,2),1)` yields the first argument within the second argument of `aExpr`, or `13`.
- `Arg(Arg(aExpr,2),2)` yields `Sqrt(14-15)`.
- `Arg(Arg(Arg(aExpr,2),2),1)` yields `14-15`.
- `Arg(Arg(Arg(aExpr,2),2),3)` yields `Empty()`

Here is a description of how the last example line unwraps itself:

1. The inner `Arg` statement is evaluated.

    ```
    Arg(aExpr,2)
      13 * Sqrt( 14 - 15 )
    ```
2. Then the next one is evaluated.

    ```
    Arg(Arg(aExpr,2),2)
    // this is equivalent to Arg(Expr (13 * Sqrt( 14 - 15 ) ), 2)
      Sqrt( 14 - 15 )
    ```
3. Finally, the outer `Arg` is evaluated.

    ```
    Arg(Arg(Arg(aExpr,2),2),3)
    // this is equivalent to Arg (Expr (Sqrt( 14 - 15 ) ), 3)
    ```

```
Empty()
```

There is only one element to the Sqrt expression, so a request for the third argument yields Empty(). To access the two arguments inside the Sqrt expression, try this:

```
Arg(Arg(Arg(Arg(aExpr,2),2),1),2);
    15
```

HeadName(*expressionArg*) returns the name of the head of the expression as a string. If the expression is an infix, prefix, postfix, or other special character operator, then it is returned as the functional equivalent.

The *expressionArg* can be a name holding an expression, an expression evaluated to an expression, or a literal expression quoted by Expr().

For example, if aExpr = expr(a+b);

- r = HeadName (aExpr) results in "Add".
- r = HeadName (Expr(sqrt(r))) results in "Sqrt".
- r = HeadName ({1,2,3}) results in "List".

In previous versions of JMP, other versions of Arg, Narg, Head, and HeadName were implemented, called ArgExpr, NArgExpr, HeadExpr, and HeadNameExpr, respectively. These did the same thing, but did not evaluate their argument. These forms are now deprecated and will not be documented in future versions.

Making Substitutions

Eval Insert is for the situation where you want to make substitutions, by evaluating expressions inside a character string.

With Eval Insert, you specify characters that delimit the start and end of an expression, and everything in between is evaluated and expanded.

There are two functions, one to return the result, the other to do it in-place.

```
resultString = EvalInsert( string with embedded
    expressions,startDelimiter,endDelimiter )
EvalInsertInto( string l-value with embedded
    expressions,startDelimiter,endDelimiter )
```

The delimiter is optional. The default start delimiter is "^". The default end delimiter is the start delimiter.

```
xstring = "def";
r = Eval Insert( "abc^xstring^ghi" );  // returns "abcdefghi";

r = "abc^xstring^ghi"; // in-place evaluation
Eval Insert Into( r ); // r now has "abcdefghi";
```

```
// with specified delimiter
r = Eval Insert( "abc%xstring%ghi","%" );  // returns "abcdefghi";

// with different start and end delimiters
r = Eval Insert( "abc[xstring]ghi","[","]" );  // returns "abcdefghi";
```

When a numeric value contains locale-specific formatting, include the `<<Use Locale(1)` option. The following example substitutes a comma for the decimal point based on the computer's locale setting.

```
Eval Insert( "^1.2^", <<Use Locale( 1 ) );
  1,2
```

Evaluate expressions inside lists

`Eval List` evaluates expressions inside a list and returns a list with the results:

```
x = { 1 + 2, 3 + 4 };
y = Eval List( x );   // returns y is {3,7}
```

`Eval List` is useful for loading the list of user choices returned by `Column Dialog` or `New Window` with the `Modal` argument.

Evaluate expressions inside expressions

`Eval Expr()` evaluates only the inner expressions and returns an expression that contains the results of the evaluation. By comparison, `Eval` evaluates the inner expressions, takes the results, and then evaluates it again.

Suppose that a data table contains a column named X3. Here is an example of using `Eval Expr()` to evaluate the inner expression first:

```
x = Expr( Distribution( Column( Expr("X"||Char( i ) ) ) ) );
i = 3;
y = Eval Expr( x );   // returns Distribution( Column( "X3" ) )
```

To evaluate further, you need to either call the result in a subsequent step, or else put `Eval()` around the `Eval Expr()`. The following examples create a Distribution report.

```
// two-step method
x = Expr( Distribution( Column( Expr( "X" || Char( i ) ) ) ) );
i = 3;
y = Eval Expr( x );
y;

// one-step method
x = Expr( Distribution( Column( Expr( "X" || Char( i ) ) ) ) );
i = 3;
Eval( Eval Expr( x ) );
```

See Table 8.3 on page 263 to learn what would happen if you tried to use Eval directly on *x* without first doing Eval Expr.

Parsing Strings into Expressions, and Vice Versa

Parsing is the syntactic scanning of character strings into language expressions. Suppose that you have read in a valid JSL expression into a character string, and now want to evaluate it. The Parse function returns the expression. To evaluate it, use the Eval function.

```
x = Parse( "a=1" ) ;  // x now has the expression a=1
Eval( Parse( "a=1" ) ); // a now has the value 1
```

To go in the reverse, use the Char function, which converts an expression into a character string. Usually the argument to a Char function is an Expr function (or a NameExpr of a variable), since Char evaluates its argument before deparsing it.

```
y = Char( Expr( a = 1 ) ); // results in y having the character value "a=1"
z = Char( 42 ); // returns "42"
```

The Char function allows arguments for field width and decimal places if the argument is a number. The default is 18 for width and 99 for decimal (Best format).

```
Char( 42, 5, 2 ); // returns the character value "42.00"
```

To preserve locale-specific formatting in the numeric value, include the <<Use Locale(1) option as shown in the following example:

```
Char( 42, 5, 2, <<Use Locale(1) ); // returns the character value "42,00" in
    the French locale
```

The reverse of Char is not quite as simple. To convert a character string into an expression, you use Parse, but to convert a character string into a number value, you use Num.

```
Parse( y );
Num( z );
```

Table 8.2 Functions to Store or Evaluate Expressions

Function	Syntax	Explanation
Char	Char(Expr(*expression*)) Char(*name*)	Converts an *expression* into a character string. The expression must be quoted with Expr; otherwise its evaluation is converted to a string.
	string = char(*number*, *width*, *decimal*)	Converts a *number* into its character representation. *Width* and *decimal* are optional arguments to specify formatting; the default is 18 for width and 99 for decimal.

Table 8.2 Functions to Store or Evaluate Expressions *(Continued)*

Function	Syntax	Explanation
Eval	Eval(*x*)	Evaluates *x*, and then evaluates the result of *x* (unquoting).
Eval Expr	Eval Expr(*x*)	Returns an expression with all the expressions inside *x* evaluated.
Eval List	Eval List(*list*)	Returns a list of the evaluated expressions inside *list*.
Expr	Expr(*x*)	Returns the argument unevaluated (expression-quoting).
NameExpr	NameExpr(*x*)	Returns the unevaluated expression of *x* rather than the evaluation of *x*. NameExpr is like Expr except that if *x* is a name, NameExpr returns the unevaluated expression stored in the name rather than the unevaluated name *x*.
Num	Num("*string*")	Converts a character string into a number.
Parse	Parse("*string*")	Converts a character string into a JSL expression.

Summary

Table 8.3 compares various ways that you can use the evaluation-control functions with *x*. Assume that a data table contains a column named X3, and *x* and *i* have been assigned:

```
x = Expr( Distribution(Column( Expr("X"||Char( i ) ) ) ) ) );
i = 3;
```

Table 8.3 Functions for Controlling Evaluation

Commands and results	Explanation

`x; // or`
`Eval(x);`
Not Found in access or evaluation of 'distribution' , Bad Argument({"X" || Char(i)})
	Eval(x) and simply calling x are equivalent.
	
	Evaluates the expression distribution(column(expr("X" || Char(i)))). This results in errors. The column name is recognized as "X"||Char(i) because it is packed by the Expr() function.

`Expr(x);` *x*	Returns the expression x (packs *an additional* layer).

Table 8.3 Functions for Controlling Evaluation   *(Continued)*

Commands and results	Explanation
Name Expr(x); *Distribution(Column(Expr("X" || Char(i))))*	Returns the expression stored in *x* exactly as is: Distribution(Column(Expr("X" || Char(i)))).
y=Eval Expr(x); *Distribution(Column("X3"))*	Evaluates the inner expression but leaves the outer expression unevaluated, so that *y* is Distribution(Column("X3")).

y; //or
Eval(Eval Expr(x));
Distribution[]
	Eval(eval expr(x)) and simply calling y are equivalent.
	
	Evaluates Distribution(Column("X3")) to launch the platform.

z = Char(nameexpr(x)); *"Distribution(Column(Expr (\!"X\!" || Char(i))))"*	Quotes the entire expression as a text string, adding \!" escape characters as needed. Note that Char(x) would first attempt to evaluate *x*, producing an error and ultimately returning a quoted missing value: "."
Parse(z); *Distribution(Column(Expr("X" || Char(i))))*	Unquotes the text string and returns an expression.

a = Parse(Char(NameExpr(x)));
Eval(EvalExpr(a));
Distribution[]
	Evaluation control taken to its logical extreme.
	
	Note that you must break this into at least two steps as shown. Combining it into one giant step produces different results because the Eval Expr layer causes the Parse layer to be copied literally, not executed.
	
	<pre>Eval(
	   EvalExpr(
	      Parse(
	         Char(
	            NameExpr(x)))));
	      Distribution(Column(Expr("X" ||
	         Char(i))))</pre>

Macros

Stored expressions can serve as a macro feature. You can store a generalized action as an expression in a JSL variable, and then call that variable wherever you need that action to be performed. This example has four macros as the arguments to `If`:

```
lastStdzdThickness=expr(
   (thickness[nrow()]-col mean(thickness)) / col std dev(thickness));
continue=expr(...<script to read in more data>...);
log=expr(print("In control at "||char( long date(today()))));
break=expr(...<script to shut down process>...); limitvalue=1;

if(lastStdzdThickness<limitvalue,log;continue,break);
```

Storing the expression with `Expr` delays its evaluation until the variable is evaluated. Any variables, data points, or expressions included in that expression are evaluated on the fly when the expression is evaluated. See "Expressions" on page 253, for detailed rules for storing expressions and later evaluating them.

Manipulating Lists

The following functions manipulate lists. They can also be used to manipulate expressions, as shown in the next section, "Manipulating Expressions" on page 268. A summary of commands with explanations is in Table 8.4 on page 270.

Most of the functions have two variants, one that produces a new value, and one that works in-place directly on its arguments. Here are some examples:

```
// In-place example
A = {2, 3, 4, 2, 1, 2, 1};
A = Remove( A, 3 ); // delete the third item in the list A, storing result in A
Remove From( A, 3 ); // delete the third item in the list A, in place
Show( A ); // A = {2, 3, 2, 1, 2, 1};

// New value example
B = {2, 3, 4, 2, 1, 2, 1};
onetwo = Insert( {1}, 2 );   // {1, 2}
Insert Into( B, {1, 2}, 4 ); // put 1, 2 before the current 4th item
Show( B ); // B = {2, 3, 4, 1, 2, 2, 1, 2, 1};
```

Note: If the position is omitted in the `Insert Into` command, items are placed at the end of the list.

```
a = Shift( {1,2,3,4}, 1 ); // store the list {2,3,4,1} in a
Shift Into( a, -1 );
Show( a ); // a = {1, 2, 3, 4};
```

```
b = Reverse( a ); // b is now {4,3,2,1}
Reverse Into( a ); // a is now {4,3,2,1}
Show( a ); // a = {4, 3, 2, 1};

s = Sort List( {1,4,2,5, -7.2, Pi(), -11, cat, apple, cake} );
Show( s ); // s = {-11, -7.2, 1, 2, 4, 5, apple, cake, cat, Pi()};

c = {5, pie, 2, Pi(), -2};
Sort List Into( c );
Show( c ); // c = {-2, 2, 5, Pi(), pie};
```

In-place functions

In-place functions are those that operate on lists or expressions directly. They have From or Into in their names (for example, Remove From and Insert Into). They do not return a result; you have to show the list to see the result. The first argument for an in-place function must be an L-value. An *L-value* is an entity such as a variable whose value can be set.

```
myList = {a, b, c, d};
Insert Into( myList, 2, 3 );
Show( myList );
   myList = {a, b, 2, c, d}
```

These examples show how to use Insert Into and Remove From with nested lists:

```
a = {{1, 2, 3}, {"A", "B", "C"}};
Show( a );
  a = {{1, 2, 3}, {"A", "B", "C"}}

Insert Into( a[1], 99, 1 );
Show( a );
  a = {{99, 1, 2, 3}, {"A", "B", "C"}}

Remove From( a[1], 1 );
Show( a );
  a = {{1, 2, 3}, {"A", "B", "C"}}
```

Not in-place functions

For the *not-in-place functions*, you must either state the list directly or else quote a name that evaluates to a list. Such functions do *not* have From or Into in their names. They *return* manipulated lists or expressions without changing the original list or expression given in the first argument.

```
myNewList = Insert( {a, b, c, d}, 2, 3 );
   {a, b, 2, c, d}

oldList = {a, b, c, d};
newList = Insert( oldList, 2, 3 );
```

```
{a, b, 2, c, d}
```

Substituting

Substitute() and Substitute Into() merit further discussion. Both functions find all
matches to a pattern in a list (or expression) and replace them with another expression. Each
pattern must be a name. The arguments are evaluated before they are applied, so most of the
time you must quote them with an Expr() function.

```
Substitute( {a,b,c}, Expr( a ), 23); // returns {23,b,c}
Substitute( Expr( Sine( x ) ), Expr( x ), Expr( y ) ); // returns Sine(y)
```

To delay evaluating an argument, use NameExpr instead of Expr:

```
a = {quick, brown, fox, jumped, over, lazy, dogs};
b = Substitute( a, Expr( dogs ), Expr( cat ) );
canine = Expr( dogs );
equine = Expr( horse );
c = Substitute( a, Name Expr( canine ), Name Expr( equine ) );
Show( a, b, c );
   a = {quick,brown,fox,jumped,over,lazy,dogs}
   b = {quick,brown,fox,jumped,over,lazy,cat}
   c = {quick,brown,fox,jumped,over,lazy,horse}
```

Substitute Into does the same work, in place:

```
Substitute Into( a, Expr( dogs ), Expr( horse ) );
```

You can list multiple pattern and replacement arguments to do more than one replacement in
a single step:

```
d = Substitute( a,
   Name Expr( quick ), Name Expr( fast ),
   Name Expr( brown ), Name Expr( black ),
   Name Expr( fox ), Name Expr( wolf )
);
   {fast,black,wolf,jumped,over,lazy,dogs}
```

Note that substitutions are done repeatedly over multiple instances of the expression pattern.
For example:

```
Substitute( Expr( a + a ), Expr( a ), Expr( aaa ) );
```

results in:

```
aaa + aaa
```

Manipulating Expressions

The functions for manipulating lists can also operate on most expressions. Be sure to quote the expression with Expr(). For example:

```
Remove( Expr( A + B + C + D ), 2 ); // results in the expression A + C + D
b = Substitute( Expr( Log( 2 ) ^ 2 / 2 ), 2, 3 ); /* returns the expression
    Log( 3 ) ^ 3 / 3*/
```

As with lists, remember that the first argument for in-place functions must be an L-value. An L-value is an entity such as a variable whose value can be set. In-place functions are those that operate on lists or expressions directly. They have From or Into in their names (for example, Remove From and Insert Into). They do not return a result; you have to show the expression to see the result.

```
polynomial = Expr( a * x ^ 2 + b * x + c );
Insert Into( polynomial, Expr( d * x ^ 3 ), 1 );
Show( polynomial );
    polynomial = d * x ^ 3 + a * x ^ 2 + b * x + c
```

For the not-in-place functions, you must either state the expression directly or else quote a name that evaluates to an expression using NameExpr. Such functions do *not* have From or Into in their names. They *return* manipulated lists or expressions without changing the original list or expression given in the first argument.

```
cubic = Insert( Expr( a * x ^ 2 + b * x + c ), Expr( d * x ^ 3 ), 1 );
    d * x ^ 3 + a * x ^ 2 + b * x + c

quadratic = Expr( a * x ^ 2 + b * x + c );
cubic = Insert( Name Expr( quadratic ), Expr( d * x ^ 3 ), 1 );
    d * x ^ 3 + a * x ^ 2 + b * x + c
```

Substituting

Substituting is extremely powerful; please review the earlier discussion "Substituting" on page 267. Here are a few notes regarding substituting for expressions.

Substitute(*pattern*, *name*, *replacement*) substitutes for names in expressions

NameExpr() looks through the name but copies instead of evaluates:

```
a = Expr(
    Distribution( Column( x ), Normal Quantile Plot )
);
Show( Name Expr( a ) );
    Name Expr(a) = Distribution(Column(x), Normal Quantile Plot);
```

Substitute() evaluates all its arguments, so they must be quoted correctly:

```
b = Substitute( Name Expr( a ), Expr( x ), Expr( :weight ) );
```

```
Show( Name Expr( b ) );
   Name Expr(b) = Distribution(Column(:weight), Normal Quantile Plot);
```

`SubstituteInto()` needs an L-value, so the first argument is not quoted:

```
Substitute Into( a, Expr( x ), Expr( :weight ) );
Show( Name Expr( a ) );
   Name Expr(a) = Distribution(Column(:weight), Normal Quantile Plot);
```

`Substitute()` is useful for changing parts of expressions, such as in the following example that tests the `Is` functions:

```
data = {1, {1, 2, 3}, [1 2 3], "abc", x, x( y )};
ops = {is number, is list, is matrix, is string, is name, is expr};
m = J( N Items( data ), N Items( ops ), 0 );
test = Expr(
   m[r, c] = _op( data[r] )
);
For( r = 1, r <= N Items( data ), r++,
   For( c = 1, c <= N Items( ops ), c++,
      Eval( Substitute( Name Expr( test ), Expr( _op ), ops[c] ) )
   )
);
Show( m );
   m =
   [1 0 0 0 0 0,
   0 1 0 0 0 1,
   0 0 1 0 0 0,
   0 0 0 1 0 0,
   0 0 0 0 1 1,
   0 0 0 0 0 1];
```

You can use `SubstituteInto()` to have JMP solve quadratic equations. The following example solves $4x^2 - 9 = 0$:

```
/* FIND THE ROOTS FOR THE EQUATION:
        a*x^2 + b*x + c = 0
The quadratic formula is x=(-b + - sqrt(b^2 - 4ac))/2a.
Use a list to store both the + and - results of the +- operation */
x = {Expr(
   (-b + Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
), Expr(
   (-b - Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
)};
Substitute Into( x, Expr( a ), 4, Expr( b ), 0, Expr( c ), -9 );

// plug in the coefficients
Show( x ); // see the result of substitution
Show( Eval Expr( x ) ); // see the solution
   x = {Expr(((-0) + Sqrt(0 ^ 2 - 4 * 4 * -9)) / (2 * 4)), Expr(((-0) - Sqrt(0
      ^ 2 - 4 * 4 * -9)) / (2 * 4))};
```

    Eval Expr(x) = {1.5, -1.5};

The functions for manipulating lists and expressions are discussed in the previous section, "Manipulating Lists" on page 265, and summarized in Table 8.4.

Table 8.4 Functions for Manipulating Lists or Expressions

Function	Syntax	Explanation
Remove	*x =* Remove(*list*|*expr*) *x =* Remove(*list*|*expr*, *position*) *x =* Remove(*list*|*expr*, {*positions*}) *x =* Remove(*list*|*expr*, *position*, *n*)	Copies the *list* or *expression*, deleting the item(s) at the indicated *position*. If *position* is omitted, items are deleted from the end. *Position* can be a list of positions. An extra argument, *n*, deletes *n* items instead of just 1.
Remove From	Remove From(*list*|*expr*, *position*) Remove From(*list*|*expr*) Remove From(*list*|*expr*, *position*, *n*)	Remove items in place. The function returns the removed item(s), but you do not have to assign them to anything. The first argument must be an L-value.
Insert	*x =* Insert(*list*|*expr*, *item*, *position*) *x =* Insert(*list*|*expr*, *item*)	Inserts a new item into the *list* or *expression* at the given position. If position is not given, it is inserted at the end.
Insert Into	Insert Into(*list*|*expr*, *item*, *position*) Insert Into(*list*|*expr*, *item*)	Same as Insert, but does it in place. *List* or *expression* must be an L-value.
Shift	*x* = Shift(*list*|*expr*) *x* = Shift(*list*|*expr*, *n*)	Shift an item or *n* items from the front to the back of the *list* or *expression*. Shift items from back to front if *n* is negative.

Table 8.4 Functions for Manipulating Lists or Expressions *(Continued)*

Function	Syntax	Explanation
Shift Into	Shift Into(*list*|*expr*) Shift Into(*list*|*expr*, *n*)	Shift items in place.
Reverse	*x*=Reverse(*list*|*expr*)	Reverse the order of elements of a *list* or terms of an *expression*.
Reverse Into	Reverse Into(*list*|*expr*)	Reverse the order of elements of a *list* or terms of an *expression* in place.
Sort List	*x*=Sort List(*list*|*expr*)	Sort the elements of a *list* or the terms of an *expression*. Numbers sort low, followed by the name value of names, strings, or operators. For example 1+2 is lower than 1-2 because the name value Add sorts lower than the name value Subtract. {1,2} sorts lower than {1,3}, which sorts lower than {1,3,0}. {1000} sorts lower than {"a"}, but {a} and {"a"} sort as equal.
Sort List Into	Sort List Into(*list*|*expr*)	Sort the elements of a *list* or terms of an *expression* in place.
Sort Ascending	Sort Ascending(*list*|*matrix*)	Returns a copy of a *list* or *matrix* with the items in ascending order.
Sort Descending	Sort Descending(*list*|*matrix*)	Returns a copy of a *list* or *matrix* with the items in descending order.
Loc Sorted	Loc Sorted(*A*, *B*)	Creates a matrix of subscript positions where the values in matrix **A** match the values in matrix **B**. **A** must be a matrix sorted in ascending order.
Substitute	R = Substitute(*list*|*expr*, Expr(*pattern*), Expr(*replacement*), ...)	Finds all matches to the pattern in the *list* or *expression*, and replaces them with the replacement expression. Each pattern must be a name. The second and third arguments are evaluated before they are applied, so most of the time you must quote them with an Expr function. To delay evaluating an argument, use Name Expr instead of Expr. You can list multiple pattern-replacement pairs for multiple substitutions in one statement.

Table 8.4 Functions for Manipulating Lists or Expressions *(Continued)*

Function	Syntax	Explanation
Substitute Into	Substitute Into(*list*|*expr*, Expr(*pattern*), Expr(*replacement*), ...)	Substitute in place.

Advanced Scoping and Namespaces

Scripts that are used in production environments need to use more advanced scoping techniques to avoid collisions between scripts. JMP provides three progressively more advanced techniques:

• The Names Default To Here() function. If you have simple scripting needs, this single command might be sufficient. See "Names Default To Here" on page 272.

• Scopes that are pre-defined by JMP. See "Scoped Names" on page 276.

• Namespaces that you can create for your scripts. See "Namespaces" on page 280.

Names Default To Here

If you write production scripts, you need to insulate the script from the current user environment. Otherwise, the variables that you use might interact with variables used by the user and by other scripts. The way to do this is to keep your names in a local environment, which you can do by setting an execution mode with the statement:

    Names Default To Here(1);

Unqualified names in a script with the *Names Default To Here* mode turned on are private to that script. However, the names persist as long as the script persists, or as long as objects created by or holding the script are still active. It is recommend that all production scripts start with Names Default To Here(1) unless there is a specific reason not to do so. When the script uses an unqualified name in this mode, that name is resolved in the local namespace.

To refer to global variables, scope the name specifically as a global variable (for example, ::global_name). To refer to columns in a data table, scope with name specifically as a data table column (for example, :column_name).

Note: Names Default To Here(1) defines a mode for a particular script. It is not a global function. One script can have this mode turned on, while another script can have it turned off. The default setting is off.

In JMP 8 and earlier, the only method to the insulate scripts was to use lengthy names that were less likely to collide with names in other scripts. Using `Names Default To Here(1)` makes this technique unnecessary.

If you have simple scripting needs, `Names Default To Here(1)` might be sufficient.

Local()

`Local()` creates local scopes only in specific contexts within a script and cannot enclose a longer script with interacting functions, while `Names Default To Here(1)` creates a local scope for an entire script. In `Local()`, you list the variables that are local to the local block. Anything not explicitly listed is not local to that block.

Local Here()

`Local Here()` provides a namespace block with `Names Default to Here(1)`. Use `Local Here()` to prevent name collisions when multiple scripts are executed from the same root namespace (for example, when a script executes two button scripts that have the same variables or an included script is run from the main script).

`Local()` does not always work due to the lifetime of the local block, but `Local Here()` is persistent across the call.

Here's an example that runs an included script from a main script. The scripts define x and y variables as different values. Variables in the included script are in the Local Here namespace.

Main.jsl:

```
Names Default To Here( 1 );

x = 5;
y = 0;

Print( "Main, Before: x=" || Char( x ) || ", y=" || Char( y ) );
Include( "Inc1.jsl" );
Print( "Main, After: x=" || Char( x ) || ", y=" || Char( y ) );
```

Inc1.jsl:

```
Names Default To Here( 1 );

Local Here( // variables are local to this script
  x = 100;
  y = 100;

Print( "Inc1: x=" || Char( x ) || ", y=" || Char( y ) );
);
Print( "Inc1 outside: x=" || Char( x ) || ", y=" || Char( y ) );
```

Return:

```
"Main, Before: x=5, y=0" // defined in Main.jsl, before running Inc1.jsl
"Inc1: x=100, y=100" // defined in Inc1.jsl
"Inc1 outside: x=5, y=0" // from Inc1.jsl, referring to Main.jsl
"Main, After: x=5, y=0" // from Main.jsl, after running Inc1.jsl
```

Notice that the x and y variables in inc1.jsl don't change because they're in the Local Here namespace.

Handling Unqualified Named Variable References

The `Names Default To Here()` function determines how *unqualified* named variable references are resolved. Explicitly scoping a variable using `here:var_name` always works, whether `Names Default To Here()` is on or off. See "Scoped Names" on page 276 for more information about `here` and other scopes.

Enabling the *Names Default To Here* mode associates a scope called `Here` with an executing script. The `Here` scope contains all of the unqualified named variables that are created when they are the target of an assignment (as an L-value). In JMP 8 and earlier, these variables normally would have been placed in the `Global` scope. Using a `Here` scope keeps variables in multiple executing scripts separate from each other, avoiding name collisions and simplifying the scripting and management of variable name collisions. You can still share information using the `Global` scope.

Names Default To Here and Global Variables

Run this example script one line at a time to see how the `Names Default To Here()` function changes the resolution of variable names.

Example Script

```
a = 1;
Names Default To Here( 1 );
a = 5;
Show( global:a, a, here:a );
  global:a = 1;
  a = 5;
  here:a = 5;
```

1. Run the first line to create a global variable named *a* that holds the value 1.

2. Run the second line to turn on the *Names Default To Here* mode.

3. Run the third line to create a new variable named *a* in the local space that holds the value 5. This line does *not* change the value assigned to the global variable *a*.

4. Run the fourth line to see how scoped and unscoped variables are resolved.

The unqualified *a* is resolved to `here:a`. If `Names Default To Here()` were not on, *a* would be resolved to the global variable named *a*.

Note that if you use `::a` instead of `global:a` in the `Show()` function, your output is a little different:

```
Show(::a, a, here:a);
  a = 1;
  a = 5;
  here:a = 5;
```

Example of Using the Names Default To Here() Function

You have two scripts with the following definitions, and `Names Default To Here()` is turned *off* (the default condition) in both scripts.

Note: Both scripts must be in separate script windows for this example.

```
a = 1; // script 1
Show( a );

a = 3; // script 2
Show( a );
```

1. Run Script 1. Here is the result:

       a = 1

2. Run Script 2. Here is the result:

       a = 3

3. Run only the `show(a);` line in Script 1. Here is the result:

       a = 3

The log shows `a = 3` because variable *a* is global, and was last modified by Script 2. This is the default behavior in JMP 9 and later, and it is the only possible behavior in JMP 8 and earlier.

4. Now turn on `Names Default To Here()` in *both* scripts.

   `Names Default To Here(1);`

Note: `Names Default To Here()` is local to a particular script. It is *not* a global setting.

---

5. Run Script 1. Here is the result:

       a = 1

6. Run Script 2. Here is the result:

       a = 3

7. Run only the `Show( a );` line in Script 1. Here is the result:

       a = 1

The log shows a = 1, because a copy of variable *a* is maintained for each script.

Note: Problems using this function are generally due to the mixing of unqualified and qualified references to global variables. Always explicitly scoping a name prevents accessing an unintended variable.

Scoped Names

Specify where a name is to be resolved by using a scope in the form `scope:name` where `scope` indicates how to resolve the name. For example, `here:name` indicates that the name should be resolved locally. Using the *Names Default To Here* mode, `here:name` is equivalent to `name`. The scope instructs how to look up the name.

The syntax is to use the colon scope operator:

    scope:name

There are several types of scopes:

- Scope can be a resolution rule. For example, `here:x` means that `x` should be resolved to a name that is local to the script. `Global:x` means that `x` should be resolved to a global name.

- Scope can be a namespace reference variable. For example. `ref:a` means that `a` should be resolved within the namespace that `ref` refers to.

- Scope can be a data table reference to look up names as column names. For example, `dt:height` means that `height` should be resolved as a column in the data table that `dt` references.

- Scope can be the name of a namespace that you created. For example, `myNamespace:b` where `myNamespace` is a namespace that you created. `"myNamespace":b` is equivalent. See "Namespaces" on page 280.

Note: If you create a function, make sure you scope it to avoid conflicts with built-in functions.

Examples of Scoping Column Formulas

The following examples demonstrate how to scope columns that contain formulas. In both scripts, *x* is a global variable, local variable, and column name.

In the first script, the column name x is unscoped. the formula in the second column multiplies the value in column x by 100. In this case, The result is a column with the values 100, 200, and 300.

    ::x = 5;

```
New Table( "Test",
   New Column( "x", Values( [1, 2, 3] ) ),
   New Column( "y", Formula( 100 * x ) ),
);
```

In the following script, the formula in column y assigns 500 to x and then adds 50 to x. Each cell in the column contains the value 550.

```
::x = 5;
New Table( "Test",
   New Column( "x", Values( [1, 2, 3] ) ),
   New Column( "y", Formula( Local( {x = 500}, x + 50 ) ) ),
);
```

Predefined Scopes

JMP provides predefined that cannot be removed or replaced. Each of these scopes has specific roles, depending on its associated object.

Table 8.5 Predefined Scopes

Scope	Description
Global	Global names are shared throughout the JMP environment.
Here	Scope of the executing script.
Builtin	JMP built-in functions. For example, `Builtin:Sqrt()`. These names are shared throughout the JMP environment.
	If you over-ride a JSL function with a custom function, you can still access the built-in JSL function by using this scope.
Local	Nearest local scope. Can be nested within the user-defined functions, `Local` and `Parameter`.
Local Here	Provides a namespace block inside `Names Default to Here(1)`. Use `Local Here()` to prevent name collisions when multiple scripts are executed from the same root namespace (for example, when a script executes two button scripts that have the same variables). The argument can be any valid JSL expression.
	`Local( {Default Local}, )` does not always work due to the lifetime of the local block, but `Local Here()` is persistent across the call.
Window	Scope of the containing user-defined window. (Rare.)
Platform	Scope of the current platform. (Rare.)

Table 8.5 Predefined Scopes *(Continued)*

Scope	Description
Box	Scope of the containing context box. A context box is nested within a user-defined window. (Rare.)

Example of Using the Window Scope

This example uses the Window scope to pass information during execution. Explicitly scoping the variables x and y to this window ensures that JMP does not try to scope x and y in other contexts, such as a data table. The variables x and y are created and used solely inside the Window environment. The Window scope is similar to using Local(), but more useful because Local() is limited in the places that it can be used.

```
New Window( "Example",
    window:gx = 20;
    window:gy = 50;
    Graph Box(
        Frame Size( 200, 200 ),
        Handle(
            window:gx,
            window:gy,
            Function( {x, y},
                window:gx = x;
                window:gy = y;
            )
        );
        Circle( {0, 0}, Sqrt( window:gx * window:gx + window:gy * window:gy ) );
    );
);
```

Figure 8.1 Example of Current Window Namespace
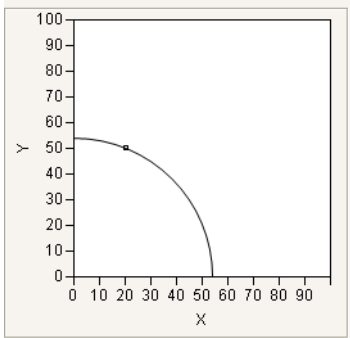
Example of Using the Here Scope

This example uses the Here scope to pass information between windows that are created by the same script. Scoping a variable using Here: is not dependent on turning Names Default To Here() on. The Here: scope is always available.

This script produces two windows and uses two different scopes.

The Launcher window asks the user for two values. Those two values are passed to the Output window, which uses them to graph a function. The Launcher window scopes aBox and bBox to the window: essentially, those two variables (pointers to Number Edit Boxes) exist only in the Launcher window and are not available to the Output window. The values from those two boxes are then copied into variables that are scoped to Here, and so are available to both windows that are produced by this script.

```
launchWin = New Window( "Launcher",
    <<Modal,
    V List Box(
        Lineup Box( N Col( 2 ), Spacing( 10 ),
            Text Box( "a" ),
            window:aBox = Number Edit Box( 50 ),
            Text Box( "b" ),
            window:bBox = Number Edit Box( 20 ),

        ),
        Lineup Box( N Col( 2 ), Spacing( 20 ),
            Button Box( "OK",
                // copy values before window goes away
                here:a = window:aBox << Get;
                here:b = window:bBox << Get;
            ),
            Button Box( "Cancel", Throw( 1 ) ) )
        ),

    )
);

New Window( "Output",
    Graph Box( Y Function( here:a + here:b * Sin( x / 30 ), x ) )
);
```

Figure 8.2 Launcher and Output
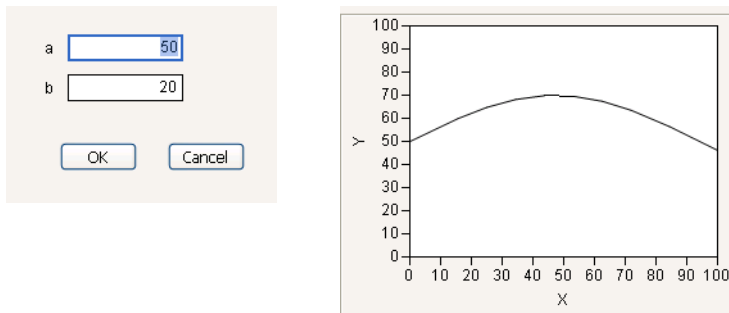
Namespaces

A *namespace* is a collection of unique names and corresponding values. You can store references to namespaces in variables. Namespace names are global, because JMP has only one namespace map. Namespace references are variables like any other variable that references an object, so they must be unique within their scope or namespace. The members of a namespace are referenced with the : scoping operator, such as my_namespace:x to refer to the object that is named x within the namespace called my_namespace. See "User-Defined Namespace Functions" on page 280 for more information about creating and managing your own namespaces. Namespaces are especially useful for avoiding name collisions between different scripts.

User-Defined Namespace Functions

Create your own namespaces to hold related sets of variables and function definitions. There are several functions that you can use to manage namespaces.

New Namespace

```
nsref = New Namespace( <"nsname">, <{ name = expr, ... }> );
```

Creates a new namespace called nsname (a string expression) and returns a reference to the namespace. All arguments are optional.

Nsname is the name of the namespace in the internal global list of namespace names. Nsname can be used as the prefix in a scoped variable. The function returns a reference to the namespace, and can also be used as the prefix in a scoped variable reference. If nsname is absent, the namespace is anonymous and is given a unique name created by JMP. Show Namespace() shows all namespaces and their names, whether assigned or anonymous.

Important: If you already have a namespace named `nsname`, it is replaced. This behavior means that while you are developing your script, you can make your changes and re-run the script without having to clear or delete your namespace. To avoid unintentional replacement, you can either use anonymous namespaces, or test to see whether a particular namespace already exists:

```
If( !Namespace Exists( "nsname" ), New Namespace( "nsname" ) );
```

A list of named expressions is optional. The names are JMP variables that exist only within the namespace.

Note: The named expressions must be a comma-separated list. Separating the expressions with semi-colons causes the list to be ignored.

These namespaces must be uniquely named to prevent collisions in situations where multiple user-defined namespaces are being used. Using anonymous namespace names prevents collisions.

Namespace

```
nsref = Namespace( "nsname" | nsref);
```

Returns a namespace reference. The argument might be either of the following:

- a quoted string that contains a namespace
- a reference to a namespace

Note: `Namespace()` returns a reference to a namespace that already exists. It does not create a new namespace.

Is Namespace

```
b = Is Namespace( nsref );
```

Returns 1 (`true`) if *nsref* is a namespace or 0 (`false`) otherwise.

As Scoped

```
b = As Scoped( "nsname", var_name);
nsname:var_name;
```

`As Scoped()` is the function form of a scoped reference. The function returns a reference to the specified variable in the specified scope.

Namespace Exists

```
b = Namespace Exists( "nsname" );
```

Returns 1 (true) if *nsname* exists in the list of global namespaces, or 0 (false) otherwise.

Show Namespaces

```
Show Namespaces();
```

Shows the contents of all namespaces contained in the list of global namespaces. Namespaces are not visible unless a reference is made to one, using either the New Namespace or Namespace functions.

Namespace Messages

In addition to the namespace management functions, a namespace also supports a set of messages to access and manipulate its contents.

Note that these messages, as with all message, must be sent to a scriptable object. A namespace name is not a defined scriptable object and cannot be used in a Send operation. However, you can use the name of a namespace in variable references. For example, nsmane::var is equivalent to nsref::var.

See the *JSL Syntax Reference* for descriptions of the messages that are supported by user-defined namespace references.

Using Namespace References

The following are all equivalent references to a variable that is named b in the namespace that is named nsname that has a reference nsref:

```
nsref:b
nsname:b
"nsname":b
nsref["b"]
nsref<<Get Value("b") // used as an r-value
```

Namespaces and Included Scripts

An included script runs in the namespace of the parent script. If the included script has its own namespace definitions, you need to do one of the following:

• manage the namespace names to avoid name collisions

• use anonymous names created by the New Namespace function

In either case, you still need to manage variable references to namespaces.

There is also an option for the Include function (New Context) that creates a namespace that the included script runs in. This namespace is anonymous and it is independent from the parent script's namespace. For example:

```
Include("file.jsl", <<New Context);
```

This anonymous namespace can be referenced using Here.

If both the parent and included scripts use the global namespace, add Names Default To Here to the New Context option to avoid name collisions. For example:

```
Include("file.jsl", <<New Context, <<Names Default To Here);
```

See "Includes" on page 294 for more information about the Include function.

Examples of User-Defined Namespaces

Creating and Using a Basic Namespace with Expressions

This example shows creating an anonymous namespace and using functions and variables within it.

```
new_emp = New Namespace(
   {name_string = "Hello, *NAME*!",

   print_greeting = Function( {a},
      Print( Substitute( new_emp:name_string, "*NAME*", Char( a ) ) )
   )}
);
```

Note that you must use the fully qualified name for variables defined within the namespace.

```
new_emp:print_greeting( 6 );
   "Hello, 6!"
```

Complex Number Operations

This example creates a namespace that contains functions to support using two-element lists to represent complex numbers, and then locks the namespace.

```
If( !Namespace Exists( "complex" ),
   New Namespace(
      "complex"
   );
   complex:makec = Function( {a, b},
      Eval List( {a, b} )
   );
   complex:addc = Function( {a, b}, a + b );
   complex:subtractc = Function( {a, b}, a - b );
   complex:multiplyc = Function( {a, b},
```

```
        Eval List( {a[1] :* b[1] - a[2] :* b[2], a[1] :* b[2] + a[2] :* b[1]} )
    );
    complex:dividec = Function( {a, b},
        d = b[1] ^ 2 + b[2] ^ 2;
        Eval List(
            {a[1] :* b[1] - a[2] :* b[2] / d, a[2] :* b[1] - a[1] :* b[2] / d}
        );
    );
    complex:charc = Function( {a},
        Char( a[1] ) || "+" || Char( a[2] ) || "i"
    );
);
Namespace( "complex" ) << Lock;
```

Here are examples using functions that are within the above user-defined namespace.

```
c1 = complex:makec( 3, 4 );
c2 = complex:makec( 5, 6 );
cadd = complex:addc( c1, c2 ); // returns {8, 10}
csum = complex:subtractc( c1, c2 ); // returns {-2, -2}
cmul = complex:multiplyc( c1, c2 ); // returns {-9, 38}
cdiv = complex:dividec( c1, c2 ); // returns {14.6065573770492,
    19.7049180327869}
Show( complex:charc( c1 ) ); // returns complex:char(c1) = "3+4i";
cm1 = complex:makec( [1, 2, 3], [4, 5, 6] ); // returns {[1, 2, 3], [4, 5, 6]}
```

Referencing Namespaces and Scopes

There are a number of factors in resolving a named variable reference. Table 8.6 describes the named variable references that are resolved for specific situations.

Notes:

- In JMP 9 and later, a, :a, and ::a have the same meaning with the *Names Default To Here* mode turned off.

- If the current point of execution is in a user-defined function, or a Local or Parameter JSL function body, then the Local namespace is used for creation when Default Local is specified. However, the reference rule is unchanged.

Table 8.6 Namespace References

Form	Reference Type	Reference Rule	Creation Rule
a	Unqualified	If the *Names Default To Here* mode is on, JMP looks for the variable in these locations: • Local namespace • Here namespace • current data table If the *Names Default To Here* mode is off, JMP looks for the variable in these locations: • Local namespace • Here namespace • Global namespace • current data table	• If the *Names Default To Here* mode is on, then JMP creates the variable in the Local namespace or in the Here namespace. • If the *Names Default To Here* mode is off, then JMP creates the variable in the Local namespace or in the Global namespace.
:a	Current data table	JMP looks for the variable in the current data table.	(Not applicable)
::a Global:a	Global	JMP looks for the variable in the Global namespace.	JMP creates the variable in the Global namespace.
ns:a dt:a Here:a "name":a expr:a	Qualified	JMP looks for the variable in the specified namespace. If the variable is not found, an error results.	JMP creates the variable in the specified namespace. Any previous values are replaced.
ns["a"] ns[expr]	Subscript	JMP looks for the variable in the specified namespace. If the variable is not found, an error results.	JMP creates the variable in the specified namespace. Any previous values are replaced.
Platform:a	Qualified	JMP looks for the variable in the encapsulating platform.	JMP creates the variable in the encapsulating platform.

Table 8.6 Namespace References *(Continued)*

Form	Reference Type	Reference Rule	Creation Rule
Local:a	Qualified	JMP looks for the variable within any nested local function, up to and including a function call boundary. See "Example of Local:a" on page 287.	JMP creates the variable in the innermost nested local function or function call boundary.
Window:a	Qualified	JMP looks for the variable in the encapsulating New Window window namespace.	JMP creates the variable in the encapsulating New Window window namespace.
Box:a	Qualified	JMP looks for the variable in the encapsulating Context Box namespace contained in a New Window window.	JMP creates the variable in the encapsulating Context Box namespace contained in a New Window window.

Example of Local:a

Sample Script	Log Output
| ```jsl
Delete Symbols();
Local( {d1l1 = 12},
   local:f1f1 = Function( {fa1, fa2},
      {fl1 = 99},
      local:fa12 = fa1 + fa2;
      Local( {d2l1 = 56},
         local:l2l2 = 78;
         Show( fa12 );
         Show( fl1 );
         Try( Show( d1l1 ), Write(
"\!n\!n***Error=" || Char( exception_msg
) || "\!n" ) );
         Show Symbols();
      );
      local:fa12;
   );
   f1f1( 2, 3 );
);
``` | ```
fa12 = 5;
fl1 = 99;
di1l = 12;

// Local

d2l1 = 56;
l2l2 = 78;

// 2 Local

// Local

fa1 = 2;
fa12 = 5;
fa2 = 3;
fl1 = 99;

// 4 Local

// Local

d1l1 = 12;

// 1 Local

5
``` |

Resolving Named Variable References

When variables are referenced within a JMP script, JMP resolves the storage location of the variable using a specific set of rules. If the variable is referenced by a qualified name, then the resolution is based on the specific qualification specification. If the variable is referenced by an unqualified name, the situation is a bit more complex. JMP looks through a hierarchy of scopes representing the point of execution with the executing script. This section describes the rules that are used to resolve named variable references.

By default, variable name resolution in JMP 9 and later worked the same way as in JMP 8 and earlier, allowing your current JSL scripts to be executed unchanged. For JMP 9 and later, the difference between qualified and unqualified variable named references is important to understand.

Qualified Named References

A qualified named reference uses the : and :: operators to provide specific information about where a referenced variable resides, or where it is created. Examples of qualified named references include the following:

```
:var
::globalvar
datatable:var
nsref:var
"nsname":var
```

Unqualified Named References

An unqualified named reference provides no explicit information to completely identify where a variable resides or where it is created. No scoping operator (: or ::) is specified in the reference. To change the behavior of JMP when resolving unqualified named variable references, use the Names Default To Here(1) function. For more information about variable name resolution, see the "Rules for Name Resolution" on page 111 in the "JSL Building Blocks" chapter.

Rules for Resolving Variable References

JMP applies rules in sequence to resolve variable references. See "Rules for Resolving Names" on page 112 in the "JSL Building Blocks" chapter.

Best Practices for Advanced Scripting

Minimize Polluting the Global Namespace and Prevent Scripts from Interacting

Always start your script with this line:

```
Names Default To Here(1);
```

Share Variables Across Scripts

Use named namespaces. Namespace names are placed in the global scope.

Use Anonymous Namespaces

Using namespace references to anonymous namespaces avoids possible conflicts with other namespaces.

Advanced Programming Concepts

This section covers some more advanced programming techniques that can be useful for developing scripts.

- "Throw and Catch Exceptions"
- "Functions"
- "Recursion"
- "Includes"
- "Load and Save Text Files"
- "Run External Programs"
- "Classes"

Throw and Catch Exceptions

A script can stop itself by executing the Throw() function. If you want to escape from part of a script when it is in an error condition, you can enclose it in a Try() expression.

If you include a character-valued expression as a Throw() argument, one of the following occurs:

- Throwing stores the string in a global variable named exception_msg. This is illustrated in the first example below.
- Throwing stores the string in a local variable named exception_msg if you define exception_msg in the Here namespace using Names Default to Here(1).

Try() takes two expression arguments. It starts by evaluating the first expression, and if or when the first expression throws an exception by evaluating Throw, it does the following:

1. Immediately stops evaluating that first expression.
2. Returns nothing
3. Evaluates the second expression.

Examples

You can use Try() and Throw() to catch an exception that JMP itself throws.

```
Try( Throw( "Hello." ), Show( exception_msg ) );
   exception_msg = "Hello.";
```

The following example prints a message to the log when the Try() expression cannot be executed:

```
Try(
```

```
    dt = Open( "Mydata.jmp" ); // a file that cannot be opened
    Summarize( a = by( age ), c = count, meanHt = Mean( Height ) );
    Show( a, c, meanHt );
,
    Print( "This script does not work without the data set." );
    Throw();
);
```

If the `Throw()` string begins with "!" and is inside a `Try()` expression, throwing creates an error message about where the exception was caught.

```
Try(
    Throw( "!This message has an exclamation point." );
,
    Write( "\!Nexception_msg: ", exception_msg );
    Print( "Hello from the catch block (WITH exclamation)." );
);
Print( "This AFTER message will NOT print due to error." );
```

When writing the `Try()` error handler code, consider wrapping the exception variable `exception_msg` in a `Char()` function. If your script calls `Concat()` without wrapping it in a `Char()` call, you might get an error depending on what was actually returned in this variable. Though some commands return a string in `exception_msg`, other commands (such as `Open()`) return a list or another data structure. This would cause an error in the error handler if it were passed directly as an argument to `Concat()`.

```
Try(
    Open( "no file" ),
    If( Is Empty( exception_msg ),
        // use Write to print "no error"if exception_msg is empty
        Write( "\!Nexception_msg (0): no error" ),
        // use Print to print the exception message
        Print( "\!Nexception_msg (1): ", exception_msg );
        /* or first make sure that exception message is a string
        and then use Concat to build the message that is passed to Print(). */
        Print( "\!Nexception_msg (2): " || Char( exception_msg ) );
    )
);
    exception_msg (1): "
    {"Cannot open table"(1, 2, "Open", Open /*###*/("no file"))}
    "
    exception_msg (2): {\!"Cannot open table\!"(1, 2, \!"Open\!", Open
        /*###*/(\!"no file\!"))}"
```

You can also use `Try()` and `Throw()` to escape from deep inside `For` loops.

```
a = [1 2 3, 4 5 ., 7 8 9];
b = a;
nr = N Row( a );
```

```
nc = N Col( a );
// a[2, 3] = 2; // uncomment this line to see the "Missing b" outcome

Try(
   sum = 0;
   For( i = 1, i <= nr, i++,
      For( j = 1, j <= nc, j++,
         za = a[i, j];
         If( Is Missing( za ),
            Throw( "Missing a" )
         );
         zb = b[j, i];
         If( Is Missing( zb ),
            Throw( "Missing b" )
         );
         sum += za * zb;
      )
   );
,
   Show( i, j, exception_msg );
   Throw();
);
   i = 2;
   j = 3;
   exception_msg = "Missing a";
```

You do not have to use Try() to make use of Throw(). In this example, Throw() is not caught by Try() but still stops a script that cannot proceed:

```
dt = New Table(); // to get an empty data table
If( N Row( dt ) == 0,
   Throw( "!Empty Data Table" )
);
```

Functions

JSL also has a function called Function to extend the macro concept with a local context arguments. Suppose that you want to create a function that takes the square root but tolerates negative arguments, returning zero rather than errors. You first specify the local arguments in a list with braces {} and then state the expression directly. You do not need to enclose the expression in Expr because Function stores it as an expression implicitly.

```
myRoot = Function( {x},
   If( x > 0, Sqrt( x ), 0 )
);
a = myRoot( 4 );  // returns 2
b = myRoot( -1 ); // returns 0
```

Functions are stored in variables, the same as values. This means that you cannot have both a
root function and a root value. It also means that you can redefine a function anytime except
when you are inside the function itself.

When a function is called, its arguments are evaluated and given to the local variables
specified in the list forming the first argument. Then the body of the function, the second
argument, is evaluated.

The values of the arguments are for the temporary use of the function. When the function is
exited, the values are discarded. The only value returned is the return value. If you want to
return several values, then return a list instead of a single value.

In defined functions, the stored function is not accessible directly, even by the Name Expr
command. If you need to access the function expression in your script, you have to create the
function within an expr() clause. For example,

```
makeFunction = Expr(
   myRoot = Function( {x},
      If( x > 0, Sqrt( x ), 0 )
   )
);
d = Substitute( Name Expr( MakeFunction ), Expr( x ), Expr( y ) );
Show( d );
makeFunction;
```

Optional Arguments

You can create optional arguments. For example:

```
f1 = Function( {x, y, z}); // all arguments are required
f2 = Function( {x, y=2, z=4}); // x is required, y and z are optional
```

Notes:

- To make an argument optional, give it a default value in the function's definition (x=1
  rather than just x).

- Optional arguments always must follow a required argument. For example, this is not
  allowed:

  ```
  Function( {x = 1, y}, ... )
  ```

  It will work when defined, but you will always have to pass in a value for x and y, because
  the default value will never be used.

- When you call the function, you cannot skip optional arguments. In this example, you
  have to supply a value for y to supply a value for z when you call the function:

  ```
  ex = Function( {x, y = 2, z = 3},
     Return( x + y + z )
  );
  ex( 1, 4 ); // passes in 1 for x and 4 for y; z will be 3. returns 8.
  ```

Local Symbols

You can declare variables as local to a function so that they do not affect the global symbol space. This is particularly useful for recursive functions, which need to keep separate the values of the local variables at each level of function call evaluation.

As shown above, a function definition looks like this:

```
functionName=Function({arg1, ...}, body);
```

You can also have the function definition default all the unscoped names to be local.

```
functionName=Function({arg1, ...}, {Default Local}, body);
```

For the purposes of creation, the use of `Default Local` localizes all the names that meet the following qualifications:

- Are not scoped as globals (for example, `::name`)
- Are not scoped as data table column names (for example, `:name`)
- Occur without parentheses after them (for example, are not of the form `name(...)`)

For example, the following function sums three numbers.

```
add3 = Function( {a, b, c},
  {temp},
  temp = a + b;
  temp + c;
);
X = add3( 1, 5, 9 );
  15
```

The following function does the same thing, automatically finding locals.

```
add3 = Function( {a, b, c},
  {Default Local},
  temp = a + b;
  temp + c;
);
X = add3( 1, 5, 9 );
  15
```

In both cases, the variable `temp` is not a global, or, if it is already a global, remains untouched by evaluating the functions.

Using `Default Local` in user-defined functions can cause some confusion because it is context-sensitive. That is, the same function may behave differently in different contexts (depending on whether same-named outer variables are in scope) because the reference rules for unqualified names are unchanged when using `Default Local`. To make a function less context-sensitive, enumerate each and every variable that you want to be local. This reduces the confusion and the potential incorrect use of outer scope variable values.

Recursion

The Recurse() function makes a recursive call of the defining function. For example, you can make a function to calculate factorials. A factorial is the product of a number, the number minus 1, the number minus 2, and so on, down to 1.

```
myfactorial = Function( {a},
   If( a == 1,
      1,
      a * Recurse( a - 1 )
   )
);
myfactorial( 5 );
   120
```

You can define recursive calculations without using Recurse(). For example, you could replace Recurse() by myfactorial, and the script would still work. However, Recurse() offers these advantages:

- It avoids name conflicts when a local variable has the same name as the function.

- You can recurse even if the function itself has not been named (for example, assigned to a global variable, such as *myfactorial* above).

Includes

The Include() function opens a script file, parses the script in it, and executes the JSL in the specified file.

```
Include( "pathname" );
```

For example,

```
Include( "$SAMPLE_SCRIPTS/myStartupScript.jsl" );
```

There is an option to obtain the parsed expression from the file, rather than evaluating the file.

```
Include( "pathname", <<Parse Only );
```

Another named option creates a namespace that the included script runs in. This namespace is an anonymous namespace and it is independent from the parent script's namespace.

```
Include( "file.jsl", <<New Context );
```

See "Advanced Scoping and Namespaces" on page 272 for information about using namespaces with your scripts.

Note the following about included files:

- JMP files aside from JSL cannot be used.

- Other recognized file types, such as image files, SAS data sets, and Microsoft Excel files cannot be used.

- Unrecognized file types are treated as a JSL file.

- Files with the .txt extension are treated as a JSL file. A text file that contains data can be included, however an error will appear since this is not valid JSL.

Load and Save Text Files

The Load Text File() and Save Text File() commands allow manipulation of text files from JSL. Note that the paths in the following code are strings.

```
text = Load Text File( "path" );
Save Text File( "path", text );
```

You can load a text file from a Web site:

```
Load Text File( "URL", <blob> );
```

The URL is a quoted string that contains the URL for the text file. The text file is returned as a string. If you add the optional named argument *blob*, a blob is returned instead.

Run External Programs

Run Program() enables you to control a program that uses standard input and standard output. Line-mode programs read commands from stdin and write answers to stdout. cmd.exe, the Windows command prompt, is one example. You can also use it to launch GUI programs, such as Windows Notepad.

```
Run Program(
   Executable( "cmd.exe" ),
   Options( {"/c", "TaskList\!n"} ),
   Read Function( "text" ) // returns all text
);
```

The preceding script writes the Windows task list and details to the log.

Figure 8.3 Log Output

```
"
Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0         24 K
System                           4 Services                   0     64,340 K
smss.exe                       516 Services                   0      1,568 K
```

You can also print the task list directly without using cmd.exe and then import the task list into a data table.

```
blob = Run Program(
   Executable( "tasklist.exe" ),
   Read Function( "blob" )
);

Open(
   blob,
   Columns(
      Column( "Image Name", Character, "Nominal" ),
      Column(
         "PID",
         Numeric,
         "Continuous",
         Format( "Best", 10 )
      ),
      Column( "Session Name", Character, "Nominal" ),
      Column(
         "Session#",
         Numeric,
         "Continuous",
         Format( "Best", 10 )
      ),
      Column( "Mem Usage", Character, "Nominal" ),
      Omitted Column( . )
   ),
   Import Settings(
      Fixed Column Widths( 26, 9, 17, 12, 12, 63 ),
      Labels( 1 ),
      Column Names Start( 2 ),
      Data Starts( 4 )
   )
);
```

Figure 8.4 Imported Data from Run Program()

	Image Name	PID	Session Name	Session#	Mem Usage
1	System Idle Proc...	0	Services	0	8 K
2	System	4	Services	0	42,560 K
3	smss.exe	456	Services	0	1,228 K

See the *JSL Syntax Reference*.

Note: If you use Run Program() to launch a GUI program, JMP is inoperable until you close the program.

Classes

Classes enable you to create objects based on templates. You can use JSL to create a definition of a class. A class definition is a template, which contains definitions for member variables, methods, and functions. After you have defined a class in JSL, you can instantiate an object that has all of the characteristics defined in the class definition.

Use the JSL function `Define Class()` to create a JSL class definition. Use the JSL function `New Object()` to instantiate an instance of a JSL class.

The JSL function `Define Class()` has a number of arguments that define the contents of the class:

```
Define Class( "class name",
   <Base Class( "base class name", <"base class name", ...> ),>
   <Show( All( Boolean ) ) | Show( <Members( Boolean ),> <Methods( Boolean ),>
   <Functions( Boolean )> ),>
   <Assignment Statements>
);
```

Where:

- `"class name"` is the name of the class being defined.
- `"base class name"` is one or more class names that make the set of base classes that are used as the foundation of the class being created.
- `Show` defines the value that appears when a class object is written out using the `Show()`, `Print()`, or `Write()` functions. Options include:
  - `All( Boolean )` shows all members, methods, and functions defined in a class object instance. The default value for All is true. If you specify the `All` option, that setting overrides any other arguments in the `Show` option.
  - `Members( Boolean )` shows all member variables contained in a class object instance.
  - `Methods( Boolean )` shows all methods contained in a class object instance.
  - `Functions( Boolean )` shows all functions contained in a class object instance.

Note: You can specify either the `All` option or any number of the `Members`, `Methods`, and `Functions` options, but you cannot specify both options at the same time.

Note: If a class contains a `_show_` method, that method overrides the `Show` option.

- `Assignment Statements` are zero or more of the following assignment statements, separated by semicolons:
  - `Member = JSL Expression`
  - `Method Name = Method( {<argument, ...>}, body )`

```
– Function Name = Function( {<argument, ...}, {<local variable, ...>},
  body )
```

Note: The Assignment Statements can alternatively be specified as a list of JSL expressions.

There are three specialized methods that you can use in a class definition:

init   Initializes the object at the time it is instantiated. This is considered the constructor for the class. You can have only one constructor per class.

show   Defines the output of the Show(), Print(), and Write() JSL functions when an instance of the class is used as the argument.

to string   Defines the output of the Char() JSL function when an instance of the class is used as the argument.

Notes:

- All methods, member variables, and functions are public.
- Functions defined within a class cannot access method variables or methods within the class.

Example of Defining a Class

The following example defines a simple class for a pet, containing the following information: name, breed, and color.

In this example, you define the constructor to require all information to create a new Pet object. Also, you define a set of accessor methods to obtain each of the attributes of a Pet object.

```
Define Class( "Pet",
   name = ""; breed = ""; color = "";
   _init_ = Method( { name, breed, color },
      this:name = name;
      this:color = color;
      this:breed = breed;
   );
   _show_ = Method( {},
      Return( name || " is a " || color || " " || breed || "." )
   );
   Get Pet Name = Method( {}, Return( name ) )
);
```

Instantiating a Class

To instantiate an instance of a JSL class object, use `New Object()` JSL function with any of the following functions and arguments:

```
New Object( "Class Name"( constructor arguments ) );
New Object( Class Name( constructor arguments ) );
New Object( Class Reference( constructor arguments ) );
New Object( "Class Name", { constructor arguments } );
New Object( Class Name, { constructor arguments } );
New Object( Class Reference, { constructor arguments } );
```

Examples of Instantiating and Using a Class

The following example shows how to instantiate a class to create and access JSL class objects.

```
class = "Pet";
xa = New Object( Pet( "Oliver", "Schnauzer", "Black" ) );
xb = New Object( class, { "Flynn", "Cockapoo", "Brown" } );
xc = New Object( class( "Mandi", "King Charles Cavalier", "Ruby" ) );
xd = New Object( "Pet"( "Charlie", "Asian", "White" ) );
xd1 = New Object( xd( "Fred", "Siamese", "Silver" ) );
xd2 = New Object( xd, { "Sam", "German Shepard", "Black and White" } );

Show( xa );
Show( xb );
Show( xc );
Show( xd );
Show( xd1 );
Show( xd2 );
xd2 = xd;                  // returns a second reference to xd
xdclone = xd << Clone;     // copy xd to a new instance, named xdclone
xd2:name = "Jerry";
Show( xd:Get Pet Name(), xd2:Get Pet Name(), xdclone:Get Pet Name() );
```

This example creates multiple instances of the Pet class. The output of the `Show()` JSL function is determined by the `_show_` method that is defined in the Pet class definition. This example also illustrates the difference between creating a second JSL reference to a class instance and cloning a class instance. When using the `Clone` message, you are creating a new class instance that is independent of the original. However, the new instance starts out as a copy of the original.

File and Directory Operations

`Pick Directory()` and `Pick File()` enable the user to select a file or directory when the script is run. To get a list of files in a directory, use `Files In Directory()`.

- "Select a Directory or File"
- "Get a List of Filenames"
- "Connect to a Network Directory"

Select a Directory or File

You can prompt the user to select a directory using the `Pick Directory()` function. The command displays a platform-specific window in which the user selects a folder. On Windows, the optional prompt string appears at the top of the Browse window (Windows) or below "Open" on the Finder window (macOS). You can also specify the path for an initial directory that appears, and specify whether files should appear in the Pick Directory window.

```
Pick Directory ( "Select a directory." );
```

To let the user select a file, use the `Pick File()` function:

```
Pick File(
    <"prompt message">, <"initial directory"> <{filter list}>,
    <first filter>, <save flag>, <"default file">,
    <multiple>)
```

The `"prompt message"` is used as the window title. The `"initial directory"` defines which folder initially appears. If a directory is defined as an empty string, the default directory is used.

You can also define the `{filter list}` used for the `Open()` window, forcing it to show only certain file types. This list must use the following syntax:

```
{"Label1|suffix1;suffix2;suffix3", "Label2|suffix4;suffix5"}
```

Each quoted string adds an entry to the **File name** list in the `Open()` window. `Label` defines the text that is displayed for each menu option. The following list of suffixes defines the file types that are displayed if its corresponding label is selected. Note the use of `"*"` to list all files in the window.

```
Pick File(
    "Select JMP File", // prompt message
    "$SAMPLE_DATA", // initial directory
    {"JMP Files|jmp;jsl;jrn", "All Files|*"}, // file filter list
    1, // intially selected item
    0, // doesn't prompt the user to save the file
```

```
      "Analgesics.jmp" // file that is selected by default
);
```

Tip: All arguments are optional, however, they are also positional. This means that you can leave out arguments only at the end of the script. Use empty strings for the arguments that you want to omit from the beginning of the script.

The script below does not set the default directory or the default file:

```
Pick File(
   "Select JMP File",
   "", // no default directory
   {"JMP Files|jmp;jsl;jrn", "All Files|*"},
   1,
   0,
   "" // no default file
);
```

The <first filter> argument sets the default selection where n is the index for the list item. In the script above, the <first filter> is the first item in the list: "JMP Files|jmp;jsl;jrn".

If <Save Flag> is false, the Multiple argument can be added to allow the user to select multiple files using the one window:

```
Pick File(
   "Select JMP File",
   "",
   {"JMP Files|jmp;jsl;jrn", "All Files|*"},
   1,
   0,
   "",
   multiple // save flag is 0, allows multiple file selection
);
```

Get a List of Filenames

To obtain a list of filenames in a specific directory, use the Files In Directory command.

```
Files In Directory( path, <recursive( 0 | 1 )> );
```

Both filenames and subdirectory names are returned as shown in the following example:

```
Files In Directory( "$SAMPLE_DATA" );
   { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design
      Experiment", "Detergent.jmp", ... }
```

Notice that the files within the Design Experiment subdirectory are not included. And only files in the root $SAMPLE_DATA directory are listed.

To return a list of all file names, add the optional recursive(*Boolean*) argument to Files In Directory:

```
Files In Directory( "$SAMPLE_DATA", recursive( 1 ) );
    { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design
       Experiment/2x3x4 Factorial.jmp", "Design Experiment/Algorithm Data.jmp",
       ... }
```

To get the full pathnames, recurse the directories and concatenate the file paths and file names. The following example loops through each file in the $SAMPLE_DATA directory and subdirectories. The file path is concatenated to each file name.

```
names = Files In Directory( "$SAMPLE_DATA", recursive( 1 ) );
For( i = 1, i <= N Items( names ), i++,
    names[i] = Convert File Path( "$SAMPLE_DATA" ) || names[i]
);
names;
    {"C:/Program Files/SAS/JMP/16/Samples/Data/2D Gaussian Process
        Example.jmp",
    "/C:/Program Files/SAS/JMP/16/Samples/Data/Abrasion.jmp", ... }
```

The Files in Directory command accepts native and POSIX paths, as well as paths using path variables. See "Path Variables" on page 145 in the "Types of Data" chapter for more information about working with paths.

Connect to a Network Directory

On Windows, you can connect to a network directory without connecting to the shared drive first.

```
Files in Directory( "\\myserver.company.com/source/Users/Smith" );
```

On macOS, you have to mount the volume first and then connect to the directory using a combination of JSL and AppleScript. Note that you can also modify this script to work on Windows. On macOS only, the user is prompted for a password.

```
mount = Function( {server},
    {Default Local},

    If( Host is( "Mac" ),

        /* Use osascript to mount the volume.
        The user will be prompted for credentials. */
        Run Program(
            Executable( "/usr/bin/osascript" ),
            Options( {"-e", Eval Insert( "mount volume \!"smb://^server^\!"" )} ),
            Read Function( "text" )
        );
        server = Concat Items( Remove( Words( server, "/" ), 1 ), "/" );
```

```
        Eval Insert( "/Volumes/^server^" );
    ,
        // Windows can use UNC to auto-mount the server.
        Eval Insert( "\!\\!\^server^" )
    )
);


users = Mount( "myserver.company.com/Users" );
Files In Directory( Eval Insert( "^users^/Smith" ) );
```

Scripting By Groups

By group arguments are supported for these functions: ColMean(), ColStdDev(),
ColNumber(), ColNMissing(), ColMinimum(), ColMaximum().

Any number of By arguments can be specified, and you can use expressions for the By
arguments. By arguments must be used in a column formula, or in the context of
ForEachRow(). The first argument can also be a general numeric expression.

Here is an example:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Mean of height by sex", Numeric, Formula( Col Mean(
    :height, :sex ) ) );

dt << New Column( "Minimum of height by sex and age",
    Numeric,
    Formula( Col Minimum( :height, :sex, :age ) )
);

dt << Distribution( Continuous Distribution( Column( :height ) ), By( :sex )
    );

dt << Tabulate(
    Show Control Panel( 0 ),
    Add Table(
        Column Table(
            Analysis Columns( :height ),
            Statistics( Mean, N, Std Dev, Min, Max, N Missing )
        ),
        Row Table( Grouping Columns( :age, :sex ) )
    )
);
```

Encrypt and Decrypt Scripts

To add a basic level of protection to scripts, you can encrypt it so only someone who knows the password can view it; you can also require a password to run it. This is useful in situations when you want to implement controlled sharing of a script.

To encrypt a script:

1. Open the script that you want to encrypt.

2. Select **Edit > Encrypt Script**.

3. Enter a decrypt password so that the user needs a password to view the script.

4. (Optional) Enter a run password to require the user to enter a password before running the encrypted script.

   Note: The passwords must consist of single-byte characters; using a text Input Method Editor (IME) does not work.

5. Click **OK**.

6. If you entered only a decrypt password, click **Yes** to confirm that you do not want to assign a run password.

   The encrypted script opens in a new window. For example:

   ```
   //-e6.0.2
   S@FTQ;VGMUTF?J<;LS;B<=IRLXCU=BV;@NS<TW;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFO
       P=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA
       @T<HNIZ;WDN?RMJ;FR>KYAXTEPPF?;XFJJOP=RQGBIAGXOYNNZ>PLIF>SW>L>ACL<KGP;=QQTC
       EG??U<PUXLV?TRBO?J>QGWTJCFJA@BNHWLVORNNGQYPIKL<IM<>JX>@G?LJ>=;RBODH@PTKK@S
       IUE;IJOR<TUTRMTGSYRSVGOR<XK<F=IWQYE=LVZFP;AUHA?YJLL;EIT?ZJZC;*
   ```

7. Save the encrypted script.

To encrypt a column formula in a script:

Column formulas created by an encrypted script are visible by default. If you want a formula to be encrypted, follow these steps:

1. Paste just the formula into a new script window.

2. Encrypt the script by clicking **Edit > Encrypt Script**.

3. Provide a password only.

4. Copy the encrypted script.

5. Paste the encrypted script in a new script as a string argument for `JSL Encrypted()` as shown below.

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );

dt << New Column( "Encrypted",
    Numeric,
    Continuous,
    Format( "Best", 12 ),
    Formula(
        JSL Encrypted(
            "//-e6.0.2
0,
136,4112eJwFwYkRhCAMAMB+roUQcITwDcoP4fovxN3Ri0B+MFqvqXB8Z0hZdFC+dgqclJoOdwrlZ
    bFjgyPlZRuR2hp399ObhpxhQXP8
GPtft0MN6Rwz8pDnjmK1K0qq1bEG+H0xvyBb
,0,"
        )
    )
);
```

To decrypt a JSL script:

1. Open the encrypted script in JMP.

2. Select **Edit > Decrypt Script**.

3. Enter the decrypt password and click **OK**.

   The decrypted script opens in a new window.

To run an encrypted JSL script:

Note: You must know which data table the script runs on before running an encrypted script. If you do not know the name of the data table, you must decrypt the script before running it.

1. Open the encrypted script in JMP.

2. Select **Edit > Run Script**.

3. Enter the run password and click **OK**.

   The script runs:

   – If the script references a data table, you are prompted to open the data table, and then the script runs.

   – If the script requires an empty data table, you must create the table and then run the encrypted script.

Note that entering the run password runs the script, but does not show the script: you must supply the decrypt password to actually view the script.

Encryption and Global Variables

Encryption alone does not hide global variables and their values. A `Show Globals()` command displays them normally. If you want to hide global variables in an encrypted script, you can give them special names.

Any global variable whose name begins with two underscore characters (__) is hidden, and `Show Globals()` displays neither its name nor its value. For example:

```
myvar = 2;
__myvar = 5;
Show Symbols();
   // Globals
   myvar = 2;
   // 2 Global (1 Hidden)
```

This strategy works whether your script is encrypted or not.

Encrypting Scripts in Data Tables

You can also encrypt a script that is saved to a data table using the `JSL Encrypted()` or `Include( Char to Blob() )` functions.

- `JSL Encrypted()` is more straightforward, because it involves one function. You can include comments inside the encrypted script.

- `Include( Char to Blob() )` lets you include comments, but not inside the script.

Note: Column formulas in encrypted scripts are not encrypted. To encrypt them, include them in the `JSL Encrypted()` function.

Follow these steps to encrypt a data table script:

1. Place the script in a script window.

   You cannot directly encrypt a script that is already saved to a data table.

2. In the script window, select **Edit > Encrypt**.

3. Enter a decrypt password.

4. (Optional) Enter a run password to require the user to enter a password before running the script.

5. If you entered only a decrypt password, click **Yes** to confirm that you do not want to assign a run password.

   The encrypted script opens in a new script window.

6. Copy the entire encrypted script.

7. Create a new data table script or open an existing script.

8. In the script portion of the window, type one of the following functions:

```
JSL Encrypted( "" );
```

```
Include( Char to Blob( "" ) );
```

9. Paste the encrypted script inside the quotation marks in the function.

10. Click **OK**.

Figure 8.5 Example of an Encrypted Data Table Script

```
JSL Encrypted(
    "//-e6.0.2
WE@GSACGT<?CKEG=NG;B<=IRLXCU=BV;@NS<TW;LR<ZFOP=JJS>NNDA@T<V><DZA
>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA
>SU@MG;LR<ZFOP=JJS>NNDA@T<FFHG=R;GYNBNME>ZBMIB?O;G<>?;UTG@=HVYCC
@MFYCMVQEOTNWASNOEV=Y<TCELCQHW>AFY>;PGSSF<IBIU?H<YLQABWEGSQBMLHW
BJE?HUC=BTP?;?;LVTT>ZQI=IXTJ<P=IHYJIPAC?PGZH;OCCMXPOJHLIRHERSW;V
;@GZDHDNF><YCN@WLBTOFZRAILT?L;NT<OOCLLLWYAIVCSPBMVL???KK=TBZJ@KU
BD>=@;EOUFMUUFLSP<HMZWL?VLHXKK?S@WAFXC>EZODQOHREMNYKRBKAWS=PSXSQ
CRYFOPVJRBNHIVLE?BNPVNM=LO>BAT<?CQF?JNNQAUEHMM;BTU><=WFVOO<;WVKB
;RSBEXTS;QGBEKNDCANPFP;DYDHTSMQNAGVESB;TCWJKGWNBG>JBESVNKOVOBPBE
;?ZPYS;X?=VE=>YFHD;;;*"
)
```

Working with Expressions in an Encrypted Script

An encrypted script might not run the same as the unencrypted script if the script attempts to convert parts of itself from expressions back to character strings. This is necessary to keep the script encrypted. You can avoid this problem by using the expression manipulation functions such as Arg(), Head(), Substitute(), and so on. You can also use Parse() to make expressions from character strings that are part of the encrypted script.

For example, here is a script that modifies an expression the wrong way for encryption:

```
ex = Expr(
   aaa = 20;
   Show( aaa );
);
// Wrong, don't do this.
temp = (Parse( Substitute( Char( Name Expr( ex ) ), "aaa", "bbb" ) ));
temp;
```

The script fails when encrypted because Char() is unable to break the encryption of the expression stored in ex. A second example follows:

```
ex = Expr(
   aaa = 20;
   Show( aaa );
);
// Best choice if it works for you.
temp = Substitute( Name Expr( ex ), Expr( aaa ), Expr( bbb ) );
temp;
```

A third example, though not optimal, follows:

```
ch = "aaa = 20; Show(aaa);";
/* Leaves the ch variable holding unencrypted text
and Temp holds an unencrypted expression.*/
temp = Parse( Substitute( ch, "aaa", "bbb" ) );
temp;
```

You might prefer the third choice over the second choice because it looks easier to understand; be aware that you might want to clear variables like ch and Temp when you are done with them. You probably encrypted the script to keep other folks from reading it, and they will definitely be able to see the content of variables when you are done.

Additional Numeric Functions

JSL also offers several categories of operations that do not make much sense in the context of the formula editor: matrix operations and numeric derivatives of functions. Algebraic derivatives are also available.

The basic arithmetic operators can also be used with matrix arguments for matrix-wise addition, subtraction, and so on. Matrices also have a few special operators for elementwise multiplication and division, concatenation, and indexing. See "Matrices" on page 202 in the "Data Structures" chapter.

Derivatives

JSL has three internal functions (not all available in the calculator) for taking derivatives.

Derivative() takes the first derivative of an expression with respect to names you specify in the second argument. A single name might be entered as this second argument; or multiple values can be specified in a list, in other words, surrounded by braces.

Note: Derivative() is also available as an editing command inside the formula editor (calculator), located on the drop-down list in the top center of the formula editor (above the keypad). To use it, highlight a single variable in the expression (to designate which variable the derivative should be taken with respect to), then select the Derivative command from the menu. The whole formula is replaced by its derivative with respect to the highlighted name.

In scripts, the easiest way to use the function is with a single name. In this example, the mathematical notation is shown first followed by the JSL equivalent.

For $f(x) = x^3$, the first derivative is $f'(x)$ or $\frac{d}{dx}x^3 = 3x^2$.

```
result = Derivative( x ^ 3, x );
Show( result );
   result = 3 * x ^ 2
```

If you want an efficient expression to take the derivative with respect to several variables, then the variables are specified in a list. The result is a list containing a *threaded* version of the original expression, followed by expressions for the derivatives. The expression is *threaded* by inserting assignments to temporary variables of expressions that are needed in several places for the derivatives.

Here is an example involving an expression involving three variables. Listing all three variables returns the first derivatives with respect to each. The result is a list with the original expression and then the derivatives in the order requested. However, note here that JMP creates a temporary variable T#1 for storing the subexpression x^2, and then uses that subexpression subsequently to save calculations.

```
result2 = Derivative( 3 * y * x ^ 2 + z ^ 3, {x, y, z} );
Show( result2 );
   result2 = {3 * y * (T#1 = x ^ 2) + z ^ 3, 6 * x * y, 3 * T#1, 3 * z ^ 2}
```

To take second derivatives, specify the variable as a third argument. Both the second and third arguments must be lists. JMP returns a list with the original expression, the first derivative(s), and then the second derivative(s) in the order requested.

```
second = Derivative( 3 * y * x ^ 2, {x}, {x} );
Show( second );
   second = {3 * y * x ^ 2, 6 * x * y, 6 * y}
```

```
second = Derivative( 3 * y * x ^ 2, {y}, {y} );
Show( second );
   second = {3 * y * (T#1 = x ^ 2), 3 * T#1, 0}
```

```
second = Derivative( 3 * y * x ^ 2, {y}, {x} );
Show( second );
   second = {3 * y * (T#2 = x ^ 2), 3 * T#2, 6 * x}
```

Num Deriv() takes the first numeric derivative of a function with respect to the value of the first argument by calculating the function at that value and at that value plus a small delta (Δ) and dividing the difference by the delta. Num Deriv2() computes the second numeric derivative in a similar fashion. These are used internally for nonlinear modeling but are not frequently useful in JSL. Note that these functions do not differentiate using a variable, but only with respect to arguments to a function. In order to differentiate with respect to x, you have to make x one of the immediate arguments, not a symbol buried deep into the expression.

Suppose to differentiate $y = 3x^2$ at the value of $x = 3$. The *incorrect* way would be to submit the following script:

```
x = 3;
```

```
n = Num Deriv( 3 * x ^ 2 );
```

The *correct* way is to make *x* an argument in the function.

```
x = 3;
f = Function( {x}, 3 * x ^ 2 );
n = Num Deriv( f( x ), 1 );
    18.000029999854
```

Consider both the mathematical notation and the JSL equivalent for another example:

For $f(x) = x^2$, it calculates $\frac{d}{dx}x^2 = \frac{(x+\Delta)^2 - x^2}{\Delta}$. At $x_0 = 3$, $\frac{d}{dx}x^2 = 6.00001$.

```
x = 3;
y = Num Deriv( x ^ 2 ); // or equivalently: y = Num Deriv( 3 ^ 2 );
    6.0000099999513
```

And here are a few more examples:

```
x = Num Deriv( Sqrt( 7 ) ); // returns 0.188982168980445
y = Num Deriv( Normal Distribution( 1 ) ); // returns 0.241969514669371
z = Num Deriv2( Normal Distribution( 1 ) ); // returns −0.241969777547979
```

Table 8.7 Derivative functions

Function	Syntax	Explanation
Derivative	Derivative(*expr*, {*name*, ...})	Returns the derivative of the *expr* with respect to *name*. Note that the second argument can be specified in a list with braces { } or simply as a variable if there is only one. Give two lists of names to take second derivatives.
NumDeriv	NumDeriv(*expr*)	Returns the first numeric derivative of the *expr* with respect to the first argument in the expression.
NumDeriv2	NumDeriv2(*expr*)	Returns the second numeric derivative of the *expr* with respect to the first argument in the expression.

Algebraic Manipulations

JSL provides a way of algebraically unwinding an expression (essentially, solving for a variable). It is accomplished through the Invert Expr() function.

```
Invert Expr(expression, name, y)
```

where

- *expression* is the expression to be inverted, or the name of a global containing the expression
- *name* is the name inside expression to unwind the expression around
- *y* is what the expression was originally equal to

For example,

```
Invert Expr( Sqrt( log( x ) ), x, y );
```

is wound around the name x (which should appear in the expression only once), and results in

```
Exp( y ^ 2 )
```

It is performed exactly as you would when doing the algebra by hand.

$$y = \text{Sqrt}(\log(x));$$

$$y^2 = \text{Log}(x);$$

$$\text{Exp}(y^2) = x;$$

Invert Expr supports most basic operations that are invertible, and makes assumptions as necessary, such as assuming you are interested only in the positive roots, and that the trigonometric functions are in invertible areas so that the inverse functions are legal.

F, Beta, Chi-square, *t*, Gamma, and Weibull distributions are supported for the first arguments in their Distribution and Quantile functions. If it encounters an expression that it cannot convert, Invert Expr() returns Empty().

JSL provides a Simplify Expr command that takes a messy, complex formula and tries to simplify it using various algebraic rules. To use it, submit

```
result = Simplify Expr(expr(expression));
```

or

```
result = Simplify Expr(nameExpr(global));
```

For example,

```
Simplify Expr( Expr( 2 * 3 * a + b * (a + 3 - c) - a * b ) );
```

results in

```
6*a + 3*b + -1*b*c
```

Simplify Expr() also unwinds nested If expressions. For example:

```
r = Simplify Expr( Expr( If( cond1, result1, If( cond2, result2, If( cond3,
    result3, resultElse ) ) ) ) );
```

results in

```
If(cond1, result1, cond2, result2, cond3, result3, resultElse);
```

Maximize and Minimize

The Maximize() and Minimize() functions find the factor values that optimize an expression. The expression is assumed to be a continuous function of the factor values.

Here is the form of the call:

```
result = Maximize(objectiveExpression,{list of factor names},
   <<option(value))
result = Minimize(objectiveExpression,{list of factor names},
   <<option(value))
```

objectiveExpression is the expression whose value is to be optimized, and can either be the expression itself, or the name of a global containing a stored expression.

{*list of factor names*} is an expression yielding a list of names involved in *objectiveExpression*.

The name can be followed by limits that bound the permitted values, for example name(lowerBound,upperBound).

If you want to limit the values on one side, make the other side a missing value, for example:

```
{beta} // unconstrained
{beta (0,1)} // constrained between 0 and 1
{beta (.,1)} // upper limit of 1
{beta (0,.)} or {beta (0)} // lower limit of 0
```

Factor values can be either numbers or matrices.

Options available, shown with their default value, include:

```
<< Tolerance(.00000001) // convergence criterion
<< Max Iter( 250) // maximum number of iterations
<< Limits() //
```

Initial values are assumed to be already supplied the factor values before calling the function.

These functions are not expected to find global optima for functions that have multiple local optima; they are useful only for taking an initial value and moving it to either a local or global optimum.

The return value is currently the value of the objective function, if the optimization was successful, or Empty() if not.

Least Squares Example

The following example uses `Minimize` to find the least squares estimates of this exponential model, with data taken from the Nonlinear Example/US Population.jmp sample data table.

```
xx = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900,
    1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990];
yy = [3.929, 5.308, 7.239, 9.638, 12.866, 17.069, 23.191, 31.443, 39.818,
    50.155, 62.947, 75.994, 91.972, 105.71, 122.775, 131.669, 151.325, 179.323,
    203.211, 226.5, 248.7];
b0 = 3.9;
b1 = .022;
sseExpr = Expr(
    Sum( (yy - (b0 * Exp( b1 * (xx - 1790) ))) ^ 2 )
);
sse = Minimize( sseExpr, {b0, b1}, <<Tolerance( .00001 ) );
Show( b0, b1, sse );
    b0 = 13.9991388055261;
    b1 = 0.0147104409355048;
    sse = 1862.14141218875;
```

Identifying Differences Between Strings, Lines, or Sequences

The `Shortest Edit Script()` function compares two strings, lines, or sequences and returns a list of changes or a matrix that describes the changes. You might use `Shortest Edit Script()` to identify differences between two columns, lists, or matrices. The function describes *a* (not *the*) shortest list of instructions to convert sequence A into sequence B. There might be more than one shortest script.

Example of Sequences

```
t1 = New Table( "t1",
    New Column( "Column 1",
        Numeric,
        Continuous,
        Format( "Best", 12 ),
        Set Values( [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 7] )
    )
);

t2 = New Table( "t2",
    New Column( "Column 1",
        Numeric,
```

```
        Continuous,
        Format( "Best", 12 ),
        Set Values( [2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5, 6] ) )
   )
);
EditScript = Shortest Edit Script( // compares column 1 in each table
   sequences(
      N Rows( t1 ),
      N Rows( t2 ),
      Function( {a, b}, // subscripts the columns
         t1:column 1[a] == t2:column 1[b] // in data tables t1 and t2
      )
   )
);
   [-1 1 . 1, // delete 1 item at position 1 in string a
    0 2 1 4, // keep 4 items at position 2 in string a, position 1 in string b
   -1 6 . 2, // delete 2 items at position 6 in string a
    1 . 5 4, // add 4 items at position 5 in string b
    0 8 9 5, // keep 5 items at position 8 in string a, position 9 in string b
   -1 13 . 1] // delete 1 item at position 13 in string a
```

Here are the columns in the matrix:

- Column 1: delete the items (-1), keep the common items (0), or add the items (1)

- Column 2: the position in string a

- Column 3: the position in string b

- Column 4: the number of items the instruction uses

A missing value indicates that the position isn't used in the comparison.

Example of Separators

The following example considers "@" and "$" to be separators.

```
aa = "this is$a test of@shortest$edit script lines$with several words";
bb = "this is a$test of$shortest$edit script lines with several@words";

// @ and $ separators
Shortest Edit Script( lines( aa, bb, separators( "@$" ) ) );
   {{"Remove", "this is$a test of@"}, {"Insert", "this is a$test of$"},
      {"Common",
   "shortest$"}, {"Remove", "edit script lines$with several words"}, {"Insert",
   "edit script lines with several@words"}}
```
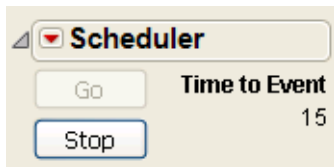
See the *JSL Syntax Reference*.

Scheduling Actions

A Schedule() function lets you set up a script to be executed some number of seconds later.

```
Schedule( 15, Print( "hello" ) );
```

Figure 8.6 JMP Scheduler

A Scheduler window shows the time until the next event and has buttons for restarting (Go) or stopping (Stop) the schedule. Its pop-up menu has a command Show Schedule, which echoes the current schedule to the log. For example, if you checked the schedule several times during the "hello" example, you would see something like this:

```
Scheduled at 11.55000000000018 :Print("hello")
Scheduled at 4.716666666666697 :Print("hello")
Scheduled at 3.083333333333485 :Print("hello")
```

The script might also be a name referring to a stored expression. For example, try submitting this script, which calls itself:

```
quickieScript = Expr(
   Show( "Hi there" );
   Schedule( 15, quickieScript );
);
quickieScript;
```

This script should show the string "Hi there" in a log window after 15 seconds, and reschedule itself for another 15 seconds, continuing until the Stop button is clicked.

More typically, in a production setting you might want to set up a schedule like this:

```
FifteenMinuteCheck = Expr(
   Show( "Checking data" );
   Open( "my file", options... );
   distribution( Column( column1 ), capability( spec limits ) );
   Schedule( 15 * 60, FifteenMinuteCheck );
);
FifteenMinuteCheck;
```

Schedule() initiates an event queue, but once it has the event queued, JMP proceeds with the next statement in the script. For example, the following has results that might surprise you:

```
Schedule( 3, Print( "one" ) );
```

```
Print( "two" );
    "two"
    "one"
```

If you want the script to wait until the scheduled events are finished before proceeding, one solution would be to use Wait( ) with a suitable pause. Another is to embed the subsequent actions into the schedule queue. Schedule() accepts a series of arguments to queue many events in sequence. Each event is a separate call to the schedule. Each event time is an absolute time relative to "now" (or the instant that **Go** is clicked). Therefore, the following sequence finishes in five seconds, not in twelve:

```
Schedule( 3, Print( "hello" ) );
Schedule( 4, Print( ", world" ) );
Schedule( 5, Print( "--bye" ) );
```

To cancel all events in a schedule queue, use Clear Schedule.

```
scheduler[1] << Clear Schedule( );
```

Note: It is not possible to create multiple threads using Schedule.

Table 8.8 Schedule commands

Message	Syntax	Explanation
Schedule	*sc*=Schedule(*n*, *script*)	Queues an event to run the *script* after *n* seconds.
Clear Schedule	*sc*<<Clear Schedule()	Cancels all events in a schedule queue.

Functions that Communicate with Users

Show(), Print(), and Write() put messages in the log window. Speak(), Caption(), Beep(), and StatusMsg() provide ways to say something to a viewer. Mail() can send an e-mail alert to a process operator.

The JMP scripting language has methods for constructing dialog boxes to ask for data column choices and other types of information. See "Modal Windows" on page 609 in the "Display Trees" chapter.

Tip: To preserve locale-specific numeric formatting in Show(), Print(), or Write() output, include <<Use Locale(1).

Writing to the Log

Show

Show() displays the items that you specify in the log. Notice that when you show variables, the resulting message is the variable name, an equal sign, and its current value.

```
X = 1;
A = "Hello, World";
Show( X, A, "foo" );
  x = 1
  a = "Hello, World"
  "foo"
```

Print

Print() sends the message that you specify to the log. Print() is the same as Show() except that it prints only the value of each variable without the variable name and an equal sign.

```
X = 1;
A = "Hello, World";
Print( X, A, "foo" );
  1
  "Hello, World"
  "foo"
```

Write

Write() sends the message that you specify to the log. Write() is the same as Print() except that it suppresses the quotation marks around the text string, and it does not start on a new line unless you include a return character yourself with the \!N escape sequence.

```
myText = "Here is a message.";
Write( "Here is a message." );
  Here is a message.

Write( myText || " Do not forget to buy milk." ); // use || to concatenate
Write( "\!NAnd bread." ); // use \!N for return
  Here is a message. Do not forget to buy milk.
  And bread.
```

The sequence \!N inserts the line breaking characters that are appropriate for the host environment. For an explanation of the three line breaking escape sequences, see "Double Quotes" on page 99 in the "JSL Building Blocks" chapter.

Send Information to the User

Beep

Beep() causes the user's computer to make an alert sound.

Speak

Speak() reads text aloud. On macOS, Speak() has one Boolean option, Wait(), to specify whether JMP should wait for speaking to finish before proceeding with the next step. The default is not to wait, and you need to issue Wait(1) each time. For example, here is a script certain to drive anybody crazy. With Wait(1), you probably want to interrupt execution before too long. If you change it to Wait(0), the iterations proceed faster than the speaking possibly can and the result sounds strange. On Windows, you can use a Wait(n) command to accomplish the same effect.

```
For( i = 99, i > 0, i--,
  Speak(
    Wait( 1 ),
    Char( i ) || " bottles of beer on the wall, " || Char( i ) || " bottles
of beer; " ||
    "If one of those bottles should happen to fall, " || Char( i - 1 ) || "
bottles of beer on the wall. "
  ) );
```

A more practical example has JMP announce the time every sixty seconds:

```
script = Expr(
  tod = Mod( Today(), In Days( 1 ) );
  hr = Floor( tod / In Hours( 1 ) );
  min = Floor( Mod( tod, In Hours( 1 ) ) / 60 );
  timeText = "time, " || Char( hr ) || " " || Char( min );
  text = Long Date( Today() ) || ", " || timeText;
  Speak( text );
  Show( text );
  Schedule( 60, script );    // seconds before next script
);
script;
```

You might use a similar technique to have JMP alert an operator that a process has gone out of control.

Caption

Caption() brings up a small window with a message to the viewer. Captions are a way to annotate demonstrations without adding superfluous objects to results windows. The first argument is an optional {h,v} screen location given in pixels from the upper left; the second argument is the text for the window. If the location argument is omitted, windows appear in the upper left corner.

You can include pauses in the playback by including the named argument Delayed and a time in seconds. Such a setting causes that caption *and all subsequent* caption windows to be delayed by that number of seconds, until a different Delayed setting is issued in a Caption statement. Use Delayed(0) to stop delaying altogether.

Specify the font type, font size, text color, or background color with the following arguments:

```
Font( font );
Font Size( size );
Text Color("color");
Back Color("color");
```

The Spoken option causes captions to be read aloud by the operating system's speech system (if available). Spoken takes a Boolean argument, and the current setting (on or off) remains in effect until switched by another Caption statement that includes a Spoken setting.

This script turns speaking on and leaves it on until the last caption. In the first caption, the font type, color, and background color is specified. Run the script and notice that the font and color settings apply only to the first caption.

```
Caption(
    {10, 30},
    "A Tour of the JMP Analyses",
    Font( "Arial Black" ),
    Font Size( 16 ),
    Text Color( "blue" ),
    Back Color( "yellow" ),
    Spoken( 1 ),
    Delayed( 5 )
);
Caption( "Open a data table." );
bigClass = Open( "$SAMPLE_DATA/Big Class.jmp" );
Caption( "A data table consists of rows and columns of data." );
Caption( "The rows are numbered and the columns are named." );
Caption( {250, 50}, "The data itself is in the grid on the right" );
Caption(
    {5, 30},
    Spoken( 0 ),
    "A panel along the left side shows columns and other attributes."
);
```

Each new `Caption` hides the previous one. In other words, there is only one caption window available at a time. To close a caption without displaying a new one, use the named argument `Remove`.

```
Caption( remove );
```

StatusMsg

This command sends a message to the status bar.

```
StatusMsg( "string" );
```

Mail

`Mail()` sends an e-mail message to a user. For example, a process control manager might include a test alert script in a control chart to trigger an e-mail warning to her pager:

```
Mail(
    "JaneDoe@company.com",
    "out of control",
    "Process 12A out of control at " || Format( Today(), "d/m/y h:m:s" )
);
```

`Mail()` can also send an attachment with the e-mail. An optional fourth argument specifies the attachment. The attachment is transferred in binary format after its existence on the disk is verified. For example, to attach the Big Class.jmp data table, submit

```
Mail(
    "JohnDoe@company.com",
    "Interesting Data Set",
    "Have a look at this class data.",
    "$SAMPLE_DATA\Big Class.jmp" );
```

Note: On macOS, `Mail()` works on Mountain Lion and later operating systems. On Mountain Lion, you must enter the e-mail address and subject in the e-mail due to operating system limitations. Click the **Send message** button to send the e-mail.

Data Tables

Working with Data Table Objects

Before you can work with a data table or with objects in a data table, you must first open the data table and assign a *reference* to it. In the *Scripting Guide*, *dt* represents a reference to a data table object.

To manipulate an object in JSL, you send a message to the reference that represents the object, asking the object to perform one of its tasks. *Messages* are commands that can be understood only in context by a particular type of object. For example, messages for data table objects include Save, New Column, Sort, and so on. Most of these messages would not make sense for another object, such as a platform object.

In this chapter, you can learn the following:

- How to assign a reference to a data table
- How to send messages to the reference, resulting in specific actions
- About the different types of messages that data table objects can understand

Most of this chapter focuses on the different types of messages that you can send to data table objects, such as data tables, columns, rows, and values.

Tip: This chapter contains most, but not all of the JSL commands that you can use with data tables. For an exhaustive list, see the JMP Scripting Index. Select **Help > Scripting Index**. Select **Objects** from the menu, and then select **Data Table**.

Contents

Get Started with Data Tables

Tip: Keep the log window open to see the output of each script that you run. Select **View > Log** to open the Log window. See "Work with the Log" on page 70 in the "Scripting Tools" chapter.

The typical way to work with values in a data table is to follow these steps:

1. Set up the data table whose values you want to access as the current data table. Or, if you already have a data table reference, you can simply use that reference.

2. Specify the row or rows whose values you want to access and specify the column name that contains the values that you want to access.

The following example opens the Big Class.jmp sample data table (making it the current data table), and then specifies row 2 in the weight column. A value of 123 is returned in the log, which is the weight for Louise in row 2.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:weight[2];
    123
```

If the data table that you want to work with is already open, proceed using one of the following examples:

```
dt = Data Table( "My Table" ); // the open table named My Table
dt = Current Data Table(); // the table in the active window
dt = Data Table( 3 ); // the third open table
```

Once you have an open data table with a reference, you can send it messages using either the << operator or the Send function. The following example illustrates both methods:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Save();
Send( dt, Save() );
```

Note the following about messages:

- You can send messages to any data table object (a data table, column, row, and so on).

- Messages sent to data table objects always have the same pattern:

```
Reference << Message( Arguments );
```

- Usually, creating a reference and sending many messages to the reference is the easiest approach. However, you can also use the direct route instead of using a reference if you have only one message to send. The following example saves the current data table:

```
Current Data Table() << Save();
```

- You can stack up a series of messages in one statement. Commands are evaluated from left to right, and each returns a reference to the affected object. The following example creates a new data table called My Table, adds two columns to it, and prompts you to save it:

```
dt = New Table( "My Table" );
dt << New Column( "Column 1" ) << New Column( "Column 2" ) << Save( "" );
```

In this example, each message returns a reference to the data table (dt).

- If you specify too few arguments for a message, JMP presents a window to get the necessary information from you. JMP often presents windows when your script is incomplete, a behavior that you can use to your advantage when writing scripts that need to query users for their choices.

- Some messages come in pairs; one to "set" or assign each attribute, and one to "get" or query the current setting of each attribute.

Why are Some Commands Sent to Objects and Others Used Directly?

New Table and Open are commands to create objects that do not exist yet. Once created, you send them messages requesting changes. To close such objects, you must close the objects' *container*, because the objects cannot delete themselves.

The following example creates a table and assigns the data table reference to the *dt* variable. New Column messages are sent to the data table reference. To delete one of those columns, the Delete Columns message is sent to the data table reference, not to the column itself.

```
dt = New Table( "Airline Data" );
dt << New Column( "Date" );
dt << New Column( "Airline" );
dt << Delete Columns( "Date" );
```

How Can I See All of the Messages that Can be Sent to a Data Table Object?

To see all of the messages that can be sent to a data table object, see the Scripting Index:

1. Select **Help > Scripting Index**.
2. Select **All Categories** from the list.
3. Select **Data Table** in the list.

You can also use the Show Properties() command. This is a good approach if you want to print or copy the list of messages. The Show Properties() command lists the messages that you can send to a data table in the Log window. Show Properties() is a command that takes any scriptable object (such as a data table or column) as its argument. To show properties for a data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show Properties( dt );
```

The resulting message list is hierarchical.

[Subtable] refers to a set of JMP menu commands. For example, the Tables subtable represents the JMP Tables menu, and the indented messages in this subtable correspond to commands in the Tables menu. Many messages include a short description.

Messages labeled as [Action] all result in some action being taken. Some messages are available for [Scripting Only]. Some messages take a Boolean argument, labeled as [Boolean].

The platforms in the Analyze and Graph menus appear in the properties list because you can send a platform name as a message to a data table. This launches the platform for the data table through the usual launch window. For more information about writing platform-specific scripts, see the "Scripting Platforms" chapter on page 469.

```
dt << Distribution( Y( height ) );
```

Note: In addition to data tables, Show Properties also works with platforms and display boxes.

The JMP Scripting Index provides more information about these properties. You can also run and modify sample scripts from the Scripting Index. Select **Help > Scripting Index** and search for the property in the Objects list.

Basic Data Table Scripting

Before you can work with data table objects, you must open or create a data table, and assign a reference to the data table.

- "Open a Data Table"
- "Create a New Data Table"
- "Import Data"
- "Set the Current Data Table"
- "Name a Data Table"
- "Save a Data Table"
- "Hide a Data Table"
- "Print a Data Table"
- "Resize a Data Table"
- "Close a Data Table"
- "Set and Get a Data Table"
- "Perform Actions on All Open Data Tables"
- "Create Journals"
- "Create Excel Workbooks"
- "Prevent Changes to a Data Table"

Open a Data Table

Use the Open() function to open a data table.

- To simply open a data table without returning a reference to it:

```
Open( "$SAMPLE_DATA/Big Class.jmp" ); // open the data table
```

- To open a data table and retain a reference to it:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" ); // open and store a reference
```

The path to the data table can be a quoted literal path (absolute or relative) or an unquoted expression that yields a pathname. Relative paths are interpreted relative to the location of the .jsl file (for a saved script). For unsaved scripts, the path is relative to your primary partition (Windows) or your *<username>*/Documents folder (macOS).

```
Open( "../My Data/Repairs.jmp" ); // relative path on Windows and macOS
Open( "::My Data:Repairs.jmp" ); // relative path on macOS
Open( "C:/My Data/Repairs.jmp" ); // absolute path
```

JMP provides shortcuts (*path variables*) to directories or files. Instead of entering the entire path to the directory or file, you include a path variable in the Open() expression. For example, JMP sample scripts typically use the $SAMPLE_DATA path variable to open files in the Samples/Data folder. For more information about path variables, see in the "Types of Data" chapter.

If you do not want to specify the entire path every time you open a data table, define a filepath string and concatenate the path with the filename:

```
myPath = "C:/My Data/Store25/Maintenance/Expenses/";
Open( myPath || "Repairs.jmp" );
```

Upon opening a data table, JMP stores the data table in memory. This means that if a script attempts to open a data table that is already open, the opened version appears. JMP does not read the version that is saved on your computer.

Test for an Open Data Table

In a script that depends on an opened data table, you can test to see whether the table is open using Is Empty() or Is Scriptable(). In the following example, the script performs a Bivariate analysis on Big Class.jmp and then closes the data table. Before proceeding to the Oneway analysis later in the script, Is Scriptable() tests for the open data table. If 1 (true) is returned, the table opens, and the script continues.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
obj = dt << Bivariate( Y( :height ), X( :age ), Fit Line );
Close( dt );
If( Not( Is Scriptable( dt ) ),
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" ),
);
obj = dt << Oneway( Y( :height ), X( :age ), Means( 1 ), Mean Diamonds( 1 ) );
```

You might want to see if the data table is open and then close it to discard any changes before running another script on the data table. To test for and close Big Class.jmp, run the following script:

```
Try( Close( Data Table( "Big Class" ) ) );
```

Prompt Users to Open a Data Table

You can use an If() expression to prompt a user to open a data table, if no open data table is found. And if they do not select a table, the script should end. The following script shows an example:

```
dt = Current Data Table();
If( Is Empty( dt ),
   Try( dt = Open(), Throw( "No data table found" ) )
);
```

The user is prompted to open a data table. If the user clicks Cancel instead of opening a data table, an error appears in the log.

Show Only Specific Columns

To open a data table and show only a specific set of columns, identify those columns in the Open() expression. This is particularly helpful with a large data table in which only a few columns are necessary.

The following example opens Big Class.jmp and includes only the age, height, and weight columns.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", Select Columns( "age", "height",
   "weight" ) );
```

You can also specify the columns to leave out of the open data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", Ignore Columns( "name", "sex" ) );
```

Create a New Data Table

You can start a new data table, or start a new data table and store its reference in a global variable. In either case, specify the table name as an argument.

```
New Table( "My Table" );
```

or

```
dt = New Table( "My Table" );
```

The following sections describe the optional arguments for the New Table() function.

Private, Invisible, and Visible

The optional visibility argument specifies whether the new data table is shown. visibility("invisible") hides the data table from view; it appears only in the JMP Home Window and the Window menu. visibility("private") avoids opening the data table. visibility("visible") shows the data table. "visible" is the default value.

```
New Table( "My Table", visibility( "private" ) );
```

A private data table enables you to avoid exposing the table to general use. Sometimes a script creates a private data table to store private data generated by a process. It is not for saving memory.

When you do not need a view of the data, or when you do not need to interact with the data, use an invisible data table.

Notes:

- The advantages of using private data tables is based on the size of the data in the table. If the data table has only five columns and 20 rows, the savings for a private data table is substantial. However, for a table with 100 columns and 1,000,000 rows, the overhead of the actual window is small in comparison to the actual data.

- Private tables are not counted when `N Table()` returns the number of open data tables.

- Private tables cannot be made current with `Current Data Table()`.

Actions

An optional argument that can define the new table. For example, the following script creates a new data table named *Little Class*. It adds three rows and two named columns as well as entering values for each cell in the table:

```
dt = New Table( "Little Class",
    Add Rows( 3 ),
    New Column( "name",
        Character,
        "Nominal",
        Set Values( {"KATIE", "LOUISE", "JANE"} )
    ),
    New Column( "height",
        "Continuous",
        Set Values( [59, 61, 55] )
    )
);
```

Import Data

Upon importing, JMP converts file types such as text files and Microsoft Excel files to the data table format. On Windows, JMP relies on filename extensions to identify the type of file and how to interpret its contents. Here are some examples of filename extensions identified by JMP:

```
Open( "$SAMPLE_IMPORT_DATA/Bigclass.xlsx" ); // Microsoft Excel file
Open( "$SAMPLE_IMPORT_DATA/Bigclass.txt" ); // text file
Open( "$SAMPLE_IMPORT_DATA/Carpoll.xpt" ); // SAS transport file
```

On macOS, JMP relies on the macOS Type and Creator codes (if present) and secondarily on three-letter filename extensions. Be sure to add the file extension before importing the file on macOS.

- Type and Creator codes are invisible data that enable the Finder to display a file with the correct icon corresponding to the application that created it.

- Files with generic icons should have the filename extensions (as with files created on other operating system).

```
Open( "$SAMPLE_IMPORT_DATA/Bigclass.txt", text );
```

Additional supported formats include .csv, .jsl, .dat, .tsv, and .jrn.

For more detailed information about import options, see the *JSL Syntax Reference*.

Import Multiple Files

When you save a script from the Multiple File Import (MFI) window into a script window, you get a compact form of the script that includes all of the parameters, default or otherwise, as messages. You could pare down the script as follows:

```
mfi = Multiple File Import(
    <<Set Folder( "$SAMPLE_IMPORT_DATA" ),
    <<Set Name Filter( "UN*.csv" ), // import files with this name
    <<Set Name Enable( 1 ),
    <<Set Add File Name Column( 1 ) // display the file names in a column
)
<<Import Data();
```

Note the slight distinction between the following expressions:

```
mfi = Multiple File Import();
```

`mfi` holds the Multiple File Import object.

```
dtlist = Multiple File Import(…) << Import Data;
```

`dtlist` holds the list of files produced by sending the `Import Data` message to the MFI object. The MFI object is gone because it was not stored in a variable or window. You can keep the MFI object by sending `Create Window` to it or storing the object in a variable, such as `mfi`, and by sending messages such as `Import Data` to that variable. Sending `Import Data` to an MFI object always returns a list, which could affect tasks such as creating reports.

`Create Window` can be useful for debugging. The window is visible *and* the MFI object is available. Run the following lines one at a time to see the results:

```
mfi = Multiple File Import();
mfi <<Create Window;
mfi <<Set Folder( "$DESKTOP" );
```

Here's an example of using a callback function to get a list of imported files after the user clicks Import on the Multiple File Import window.

```
Delete Directory( "$TEMP/test" );
Create Directory( "$TEMP/test" );
Save Text File( "$TEMP/test/argylesocks1.txt", "a1\!n1" ); // make some
Save Text File( "$TEMP/test/argylesocks2.txt", "a2\!n1" ); // data
```

```
global:importedFiles={};

mfi = Multiple File Import( // make an MFI object
    <<Set Folder( "$TEMP/test/" ),
    <<Set Name Filter( "argylesocks?.txt;" ),
    <<Set Name Enable( 1 ),
    <<Set Add File Name Column( 1 ),
    <<Set Import Callback(
        Function( {this, files}, // the callback function
            global:importedFiles = files;
/* optional: omit Close Window to keep the Multiple File Import window open so
   that the user can modify the settings. */
            win << Close Window;
        )
    )
);
win = mfi << Create Window; // open the window
While(
    Try(
        win << Parent;
        1;
    ,
        0
    ),
    Wait( 1 )
); // wait for the dialog to close
Show( global:importedFiles ); // proceed
```

Notes:

• See *Using JMP* for more information about MFI.

• See for more information about using MFI to import images from a web page into a data table.

Import a PDF File

In a PDF file, data is laid out in tables. JMP auto-detects these tables to determine where the rows and columns begin and end.

You write scripts that specify the dimensions of each table on each page, the number of column headers, whether the tables should be concatenated or individual, and more.

Note: A PDF file consists of tables that contain data. In the PDF preview of the PDF Import Wizard, the tables are outlined and selected on each page.

The following options are available for importing a PDF file:

```
Open( "PDF file path" ),
   Tables(
      Table( "data table name",
      Add Rows( Header Rows( number ),
      (Rect( left, right, top, bottom ),
      Page( number ), Row Borders(), Column Borders())) |
         Tables( Combine( All | Matching Headers | None),
            Minimum Rows( number ), Minimum Columns(#))))
      | PDF wizard );
```

Concatenate Tables Into One Data Table

This section shows how to import and concatenate the tables into one data table. The script that you create appears in a Source variable in the final data table, so you can run it again anytime.For this example, Figure 9.1 shows the sample import data file called Food Distribution.pdf, which contains one table per page.

Figure 9.1 Initial PDF File

COST OF FOOD DISTRIBUTION PROGRAMS						
(Data as of March 08, 2019)						
	Nutrition	Food Dist.			Emergency	
	Services	on Indian	Charitable	Commodity	Food	
Fiscal	Incentive	Reserv.	Institutions	Supplemental	Assistance	Total
Year	Prog. 1]	2]	3]	Food 4]	5]	6]
		--------------------------------Millions of Dollars----------------------------				
1969	--	223.9	25.4	1.0	--	250.3
1970	--	281.6	22.5	7.8	--	311.9
1971	--	308.4	24.5	12.8	--	345.7
1972	--	298.6	25.8	12.9	--	337.3
1973	--	241.2	27.4	13.3	--	281.9

```
Open(
   "$SAMPLE_IMPORT_DATA/Food Distribution.pdf",
   PDF Tables(
      Table( // import tables into one table
         Table Name( "Food Distribution" ), // data table name
         Add Rows( // rows on page 1
            Header Rows( 7 ), // number of header rows
            Page( 1 ),
            Rect( 0.6922, 0.7155, 7.2842, 10.1124 ) // size of the table
         ), // rows and specifications on page 2
         Add Rows( Page( 2 ), Rect( 0.6922, 0.7156, 7.2842, 10.0708 ) ) )
      )
   )
);
```

Tip: Import a PDF file in the PDF Import Wizard and copy the script from the Source variable into a standalone script. This is a quick way to get the settings and avoid having to guess at the `Rect` settings.

Save as Individual Data Tables

The following syntax shows how to save tables in a PDF file into individual data tables. Each resulting data table contains a Source script like the one below.

```
Open(
   "PDF file path",
   PDF Tables(
      Table(
         Table Name( "data table name" ),
         Add Rows( Page( number ), Rect( left, right, top, bottom ) )
      )
   )
);
```

Concatenate Data Based on Column Headers

The following example shows how to concatenate data when column names on each page match.

Figure 9.2 Data with Matching Column Headers on each Page

Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa

```
pdftable = Open(
   "$SAMPLE_IMPORT_DATA/Iris.pdf",
   PDF All Tables( Combine( Matching Headers ) )
);
```

Figure 9.3 Final Data Table

		Page	Table	Sepal length	Sepal width	Petal length	Petal width	Species
	1	1	1	5.1	3.5	1.4	0.2	setosa
	2	1	1	4.9	3	1.4	0.2	setosa
	3	1	1	4.7	3.2	1.3	0.2	setosa
	4	1	1	4.6	3.1	1.5	0.2	setosa
	5	1	1	5	3.6	1.4	0.2	setosa
	6	1	1	5.4	3.9	1.7	0.4	setosa
	7	1	1	4.6	3.4	1.4	0.3	setosa
	8	1	1	5	3.4	1.5	0.2	setosa
	9	1	1	4.4	2.9	1.4	0.2	setosa
	10	1	1	4.9	3.1	1.5	0.1	setosa
	11	1	1	5.4	3.7	1.5	0.2	setosa
	12	1	1	4.8	3.4	1.6	0.2	setosa
	13	1	1	4.8	3	1.4	0.1	setosa
	14	1	1	4.3	3	1.1	0.1	setosa
	15	1	1	5.8	4	1.2	0.2	setosa
	16	1	1	5.7	4.4	1.5	0.4	setosa
	17	1	1	5.4	3.9	1.3	0.4	setosa
	18	1	1	5.1	3.5	1.4	0.3	setosa

Panel at left:
- Iris Page(1) T...
  - Source
- Columns (7/0)
  - Page
  - Table
  - Sepal length
  - Sepal width
  - Petal length
  - Petal width
  - Species
- Rows
  - All rows   150
  - Selected   0
  - Excluded   0
  - Hidden   0
  - Labelled   0

Import PDF Pages that Contain Multiple Tables

A PDF file might contain data divided into multiple columns on each page. Figure 9.4 shows the sample import data PDF file called Groundhog Day Predictions.pdf, which contains multiple tables per page.

Figure 9.4 Multiple Columns on a Page

The following script imports all columns into a single data table.

```
Open(
    "$SAMPLE_IMPORT_DATA/Groundhog Day Predictions.pdf",
    PDF Tables(
        Table(
            Table Name( "Groundhog Day Predictions" ),
            // specifications for page 1
            Add Rows( Page( 1 ), Rect( 0.9726, 1.4325, 3.0351, 9.9974 ) ),
            Add Rows( Page( 1 ), Rect( 3.3063, 1.4325, 5.3688, 9.9608 ) ),
```

```
                Add Rows( Page( 1 ), Rect( 5.6399, 1.4325, 7.7025, 8.7925 ) ),
                // specifications for page 2
                Add Rows( Page( 2 ), Rect( 0.9726, 1.4325, 3.0351, 9.9441 ) ),
                Add Rows( Page( 2 ), Rect( 3.3063, 1.4325, 5.3688, 9.9274 ) ),
                Add Rows( Page( 2 ), Rect( 5.6399, 1.4325, 7.7025, 8.8225 ) ),
                // specifications for page 3
                Add Rows( Page( 3 ), Rect( 0.9726, 1.4325, 3.0351, 7.6404 ) ),
                Add Rows( Page( 3 ), Rect( 3.3063, 1.4325, 5.3688, 7.1254 ) ),
                Add Rows( Page( 3 ), Rect( 5.6399, 1.4325, 7.7025, 7.4721 ) )
            )
        )
    );
```

Figure 9.5 Final Data Table

Export Data to a Text String

You can export data from a PDF file into a text string. You might want to do this to extract a pattern from the data.

Here is a basic example that imports all text from the Iris.pdf sample import data.

Open("$SAMPLE_IMPORT_DATA/Iris.pdf", PDF text);

The following script finds the word "Figure" followed by a space, a digit, a period, and another digit and then prints the data to the log.

```
    txt = Open( "$JMP_HOME/Documentation/JMP Documentation Library.pdf",
        PDFText() );
    figures = {};
    For( page = 1, page < N Items( txt ), page++,
        // search for the string
        next = Regex Match( txt[page], "Figure\s+\d+\.\d+", "" );
        While( N Items( next ) > 0,
```

```
        figures = Insert( figures, next );
        // extract the string
        next = Regex Match( txt[page], "Figure\s+\d+\.\d+", "" );
    );
);
Print( figures );
    {"Figure 2.1", "Figure 2.2", "Figure 2.3", "Figure 2.4", "Figure 2.4",
        "Figure 2.4", "Figure 2.4", "Figure 2.5", "Figure 2.6", "Figure 2.7",
        "Figure 2.8", "Figure 2.9", "Figure 2.10", "Figure 2.10", ...}
```

Import Data from a Text File

The Import Settings in the Text Data Files preferences determine how text files are imported. For example, column names begin on line one and data begin on line two by default. To use different settings, specify the import settings as Open() options in your script.

The default Import Settings and your custom import settings are saved in the data table Source script, so you can reimport the data using the same settings. However, the default Import Settings are optional in the script.

The following Open() options are available:

```
/* Charset options: "Best Guess", "utf-8", "utf-16", "us-ascii",
    "windows-1252", "x-max-roman", "x-mac-japanese", "shift-jis", "euc-jp",
    "utf-16be", "gb2312" */
CharSet("option")
Number of Columns(Number)

/* colType is Character|Numeric
colWidth is an integer specifying the width of the column */
Columns(colName=colType(colWidth),... )
Treat Empty Columns as Numeric(Boolean)
Scan Whole File(Boolean)
End Of Field(Tab|Space|Comma|Semicolon|Other|None)
EOF Other("char")
End Of Line(CRLF|CR|LF|Semicolon|Other)
EOL Other("Char")
Strip Quotes|Strip Enclosing Quotes(Boolean)
Labels | Table Contains Column Headers(Boolean)
Year Rule | Two digit year rule ("Decade Start")
Column Names Start | Column Names are on line(Number)
Data Starts | Data Starts on Line(Number)
Lines to Read(Number)
Use Apostrophe as Quotation Mark
CompressNumericColumns(Boolean)
CompressCharacterColumns(Boolean)
CompressAllowListCheck(Boolean)
```

The following script opens a text file of comma-delimited text, which includes no column names. The script defines the column names and the column widths.

```
Open(
    "$SAMPLE_IMPORT_DATA/EOF_comma.txt",
    End of Field( comma ),
    Labels( 0 ),
    Columns(
        name = Character( 12 ),
        age = Numeric( 5 ),
        sex = Character( 5 ),
        height = Numeric( 3 ),
        weight = Numeric( 3 )
    )
);
```

Here is an example of opening a text file in which the field separator is a space and the text file does not contain column headings:

```
Open(
    "$SAMPLE_IMPORT_DATA/EOF_space.txt",
    Labels( 0 ),
    End of Field( Space )
);
```

In the preceding example of a text file with space field separators, JMP can select the best file format so that you do not have to specify the labels or separator:

```
Open(
    "$SAMPLE_IMPORT_DATA/EOF_space.txt",
    Import Settings( Guess File Format() )
);
```

To set the import options interactively, include the `Text Wizard` argument. A preview of the text file opens in the text import window.

```
Open( "$SAMPLE_IMPORT_DATA/EOF_space.txt", "Text Wizard" );
```

The following sections describe each argument in more detail. For more information about import options, see the *JSL Syntax Reference*.

Number of Columns

Specifies the total number of columns in the source file. This option is important if data is not clearly delimited.

Columns

Identifies column names, column types, and column widths with a `Columns` argument as shown in the preceding examples.

If you specify settings for a column other than the first column in the file, you must also specify settings for all the columns that precede it. Suppose that you want to open a text file that has four columns (name, sex, and age, and ID, in that order). age is a numeric column, and the width should be 5. You must also set the name and sex column types and widths, and list them in the same order:

```
Columns(
   name = Character( 15 ),
   sex = Character( 5 ),
   age = Numeric( 5 )
);
```

You are not required to provide settings for any columns that follow the one that you want to set (in this example, ID).

After the data is imported, you use the modeling type for a column. See "Set or Get Data and Modeling Types" on page 416.

Note: Most of the following arguments are defined in the JMP preferences. To override the preference, include the corresponding argument described below in your import scripts.

Treat Empty Columns as Numeric

Imports columns of missing data as numeric rather than character data. A period, Unicode dot, NaN, or a blank string are possible missing value indicators. This is a Boolean value. The default value is false.

Scan Whole File

Specifies how long JMP scans the file to determine data types for the columns. This is a Boolean value. The default value is true; the entire file is scanned until the data type is determined. To import large files, consider setting the value to false, which scans the file for five seconds.

Strip Quotes | Strip Enclosing Quotes

Specifies whether to include or remove the double quotation marks (") that surround string values. This is a Boolean value. The default value is `true`.

For example, suppose that the field delimiter is a space:

- *John Doe* is interpreted as two separate strings (*John* and *Doe*).

- "*John Doe*" is interpreted as a single string. Most programs (including JMP) read a quotation mark and ignore other field delimiters until the second quotation occurs.

- If you include `Strip Quotes(1)`, "*John Doe*" is interpreted as *John Doe* (one string without quotation marks).

Note that many word processors have a "smart quotation marks" feature that automatically converts double quotation marks (") into left and right curled quotation marks (" "). Smart quotation marks are interpreted literally as characters when the text file is imported, even when JMP strips double quotation marks.

End of Line(CR | LF | CRLF | Semicolon | Other)

Specifies the character or characters that separate rows:

- CR for carriage returns (typical for text files created on macOS up to version 9)
- LF for linefeeds (typical for UNIX and Mac OS X text files)
- CRLF for both a carriage return followed by a linefeed (typical for Windows text files).

All three characters are line delimiters by default.

Use the Other option to use an additional character for the row separator, which you must specify in the EOLOther argument. JMP interprets either this character or the default character as a row separator.

End of Field(Tab | Space | Spaces | Comma | Semicolon | Other | None)

Specifies the character or characters used to separate fields.

Notes:

- The default field delimiter is Tab.
- Use the Other option to use a different character for the field separator, which you must specify in the EOFOther argument.
- The Space option uses a single space as a delimiter.
- The Spaces option uses two or more spaces.

EOFOther, EOLOther

Specifies the character or characters used to separate fields or rows. For example, EOLOther("*") indicates that an asterisk separates rows in the text file.

Labels | Table Contains Column Headers

Indicates whether the first line of the text file contains column names. This is a Boolean value. The default value is true.

Year Rule | Two Digit Year Rule

Specifies how to import two-digit year values. If the earliest date is 1979, specify "1970". If the earliest date is 2001, specify "20xx".

Column Names Start | Column Names Are on Line

Specifies the starting line for column names. The following example specifies that the column names in the text file start on line three.

```
Open(
    "$SAMPLE_IMPORT_DATA/Animals_line3.txt",
    Columns(
        Column( "species", Character, "Nominal" ),
        Column( "subject", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "miles", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "season", Character, "Nominal" )
    ),
    Column Names Start( 3 )
);
```

Data Starts | Data Starts on Line

Specifies the starting line for data.

The following example specifies that the data in the text file start on line five.

```
Open(
    "$SAMPLE_IMPORT_DATA/Bigclass_L.txt",
    Columns(
        Column( "name", Character, "Nominal" ),
        Column( "age", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "sex", Character, "Nominal" ),
        Column( "height", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "weight", Numeric, "Continuous", Format( "Best", 10 ) )
    ),
    Data Starts( 5 )
);
```

Lines to Read

Specifies the number of lines to include in the data table. JMP starts counting after column names are read.

The following example includes only the first 10 lines in the data table.

```
Open(
    "$SAMPLE_IMPORT_DATA/Bigclass_L.txt",
    Columns(
        Column( "name", Character, "Nominal"),
        Column( "age", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "sex", Character, "Nominal" ),
        Column( "height", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "weight", Numeric, "Continuous", Format( "Best", 10 ) )
```

```
    ),
    Lines To Read( 10 )
);
```

Use Apostrophe as Quotation Mark

For data that are enclosed in apostrophes, this option treats apostrophes as quotation marks and omits them. For example, '2010' is imported as 2. This is a Boolean value. The default value is false.

This option is not recommended unless your data comes from a nonstandard source that places apostrophes around data fields rather than quotation marks.

Set Text Import Preferences

If you want to set preferences for importing text, it can be helpful to first see a list of all preferences. To do so, use the Show Preferences (All) function.

Then, copy and paste only the section you want into a Preferences function. For example, to specify import settings, write the following expression:

```
Preferences(
    Import Settings(
        End Of Field( Tab, Spaces, Comma )
    )
);
```

Import Data from a Microsoft Excel File

When you open a Microsoft Excel workbook in JMP, the file is converted to a data table. JMP supports .xls, .xlsm, and .xlsx formats. See *Using JMP* for more information about Microsoft Excel support.

Excel Preferences

In the JMP preferences, settings in the Third Party Data group can help determine how worksheets are imported:

Excel Open Method   Specifies how a Microsoft Excel file should be opened by default, when using a non-specific open statement.

   Use Excel Wizard   Opens the Excel Import Wizard to import the file. This is the default setting.

   Open All Sheets   Opens all worksheets in the Microsoft Excel file.

   Select Individual Worksheets   Prompts users to select one or more worksheets when they open the file.

Use Excel Labels as Headings Determines whether text in the first row of the worksheet is converted to column headings in the data table.

By default, JMP takes the best guess. If names have been defined for all cells in the first row, the text in those cells is converted to column heading. Otherwise, columns are named *Column 1*, *Column 2*, and so on.

To override a preference, include the corresponding argument described later in this section.

Note: See "Create Excel Workbooks" on page 361 for information about exporting data tables to an Excel workbook.

Open() Function

Using the Open() function without additional arguments to open an Excel file has different behaviors depending on the context:

- If the Open() function is a direct part of the script, the Excel files open into data tables using your Excel preferences. The following example opens both worksheets into data tables without using the wizard:

```
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
dt = Open( Path );
```

Note: To use the Excel Wizard, you must specify the Excel Wizard option in the argument as in Open("$SAMPLE_IMPORT_DATA/Team Results.xlsx", "Excel Wizard");.

- However, if the Open() function is part of a script that is run from clicking a button, the Preview window opens and requires the user to interact with it. Run the following example and click the button to see the Excel Import Wizard:

```
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
New Window( "button", Button Box( "Open", dt = Open( Path ) ) );
```

- To prevent the button script from opening the Preview window and importing the Excel file directly, provide additional arguments to the Open() function. Run the example and click the button. Both worksheets are opened into data tables without using the Excel Import Wizard.

```
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
New Window( "button", Button Box( "Open", dt = Open( Path, Use for all
    sheets(1) ) ) );
```

Alternatively, you can set the Excel Open Method preference to open all sheets:

```
Preference( Excel Open Method( "Open All Sheets" ) );
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
New Window( "button", Button Box( "Open", dt = Open( Path ) ) );
```

Open a Workbook in the Excel Import Wizard

The Excel Import Wizard shows a preview of the data and lets you modify the settings before importing the data. Specify `"Excel Wizard"` as the argument.

```
dt = Open( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", Excel Wizard );
```

Customize Import Settings

Customize settings such as the number of rows in the column headers.

```
Open(
   "$SAMPLE_IMPORT_DATA/Team Results.xlsx",
   Worksheets( "Grouped Team Results" ),
   Use for all sheets( 1 ),
   Concatenate Worksheets( 0 ),
   Create Concatenation Column( 0 ),
   Worksheet Settings(
      1,
      Has Column Headers( 1 ),
      Number of Rows in Headers( 1 ),
      // customization based on Team Results structure
      Headers Start on Row( 3 )
   )
);
```

When you click a custom button to open a workbook, the Excel Open Method preference applies. To force the worksheet to open directly, specify the Excel Open Method in your script:

```
Preference( Excel Open Method( "Open All Sheets" ) );
```

Import Specific Worksheets

Suppose that you want to import data from specific worksheets in your workbook. Specify those worksheets using the `Worksheets` argument. In the following example, the worksheet named *small* is imported into JMP.

```
Open( "C:\My Data\cars.xlsx", Worksheets( "small" ) );
```

Or specify the number of the worksheet, the third worksheet in the following example:

```
Open( "C:\My Data\cars.xlsx", Worksheets( "3" ) );
```

Import multiple or all worksheets by including the worksheet names in a list:

```
Open( "C:\My Data\cars.xlsx", Worksheets( {"small", "medium", "large"} ) );
```

Import XML and JSON Files

The XML Import Wizard imports two types of nested text data: XML and JSON files. Both kinds of files contain structured text that can be nested to represent hierarchical relationships. This wizard helps you pick out the elements of the hierarchy that hold values and elements that determine rows to build a data table.

A useful way to imagine a nested text file: the document is a book of short stories. Some of the short stories might have chapters, others only have paragraphs. Chapters have paragraphs. Paragraphs have sentences. Here's an example of the full tree for such a book. This content is also available in the JMP Samples/Import Data directory as Book.xml.

```
<book>
  <story name="car poll">
    <wheels>4</wheels>
    <para><name>chev</name></para>
    <para><name>ford</name></para>
    <para><name>volk</name></para>
  </story>
  <story name="big class">
    <para><name>ralph</name><height>6</height></para>
    <para><name>billy</name><height>5</height></para>
  </story>
  <story name="cheese" location="NC">
    <chapter name="american">
      <para>1<price>2.00</price><quantity>2</quantity></para>
      <para>2<price>2.10</price><quantity>4</quantity></para>
      <para>3<price>2.20</price><quantity>3</quantity></para>
    </chapter>
    <chapter name="swiss">
      <para>1<price>3.00</price><quantity>3</quantity></para>
      <para>2<price>3.10</price><quantity>2</quantity></para>
    </chapter>
  </story>
</book>
```

This XML document (the book) contains three short stories; each short story is a data table, all different. This particular XML file uses paragraphs to represent rows in a table in all of the tables. That is not required, nor are any of the names special. If you wanted to import the cheese table, which is slightly more complicated than the other two, you'd probably want a variety column, containing "american" or "swiss" for each row. You would also want price and quantity columns. And you might want a column that doesn't have a clearly defined name for the 1, 2, 3, 1, 2 values of <para>. You'd expect five rows. Finally, you might want a table variable or a column for the location NC value.

The following example shows how to import the preceding data into a data table.

```
Open(
```

```
            "$SAMPLE_IMPORT_DATA/Book.xml",
        XML Settings(
            Row( "/book/story/chapter/para" ),
            // create a row for each value in <para>
            Col(
                "/book/story/@location",
                // create a column for the location attribute
                Column Name( "location" ),
                // rename the column
                Fill( "Use Forever" ),
                // fill an empty cell with the preceding value
                Type( "Character" ), // data type
                Format( "best" ) // column format
            ),
            Col(
                "/book/story/chapter/@name",
                Column Name( "variety" ),
                Fill( "Use Once" ),
                Type( "Character" ),
                Format( "best" )
            ),
            Col(
                "/book/story/chapter/para",
                Column Name( "interesting" ),
                Fill( "Use Once" ),
                Type( "Numeric" ),
                Format( "best" )
            ),
            Col(
                "/book/story/chapter/para/price",
                Column Name( "price" ),
                Fill( "Use Once" ),
                Type( "Character" ),
                Format( "best" )
            ),
            Col(
                "/book/story/chapter/para/quantity",
                Column Name( "quantity" ),
                Fill( "Use Once" ),
                Type( "Character" ),
                Format( "best" )
            )
        ),
        XML Wizard( 0 ) /* omit or change to 1 to preview the data
    )                    in the XML Import Wizard */
```

Import from and Export to Google Sheets

A Google Sheet is a spreadsheet (which consists of sheets, like tabs) that is saved in your Google account. This enables you to continue to edit the spreadsheet and to share it with others.

To write a script to import data from a Google spreadsheet, log in to your Google account and get the URL for the spreadsheet. Also notice how the data is structured. For example, you must know which row the data begin on if there are column headers.

Give Google access to JMP when running the script for the first time.

To import data from a Google spreadsheet

```
dt = Google Sheet Import(
   Email( "yourgoogleaccount@gmail.com" ), // modify the email address
   // modify the sheet URL
   Spreadsheet(
   "https://docs.google.com/spreadsheets/d/1J01M4ScGsAHb-bCUaOi575WWvQjXciQIV
   NOwVTrXDSe/edit#gid=2303744702" ),
   Sheets( "May","June" ), // specify the sheet name(s)
   Sheet Settings( // specify the structure of the data
      Has Column Headers( 1 ),
      Headers Start on Row ( 1 ),
      Data Starts on Row( 2 ),
      Cell Range( "" ),
      Import Cell Colors( 1 ),
      Suppress Empty Columns( 1 )
   )
);
```

Note: See *Using JMP* for more information about security, country restrictions, and more.

To export a JMP data table to a Google spreadsheet

Export a JMP data table to a new spreadsheet in Google Sheets by following this example:

```
email = "yourgoogleaccount@gmail.com"; // modify the Google account
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Google Sheet Export(
      dt,
      Email( email ),
     New Spreadsheet( "JSL Example" ), // specify the name of the spreadsheet
      Sheet Name( "Example 1" ) // specify the sheet name
);
```

"1" is returned in the log when the export is successful.

Export a JMP data table to an existing spreadsheet in Google Sheets by following this example. You'll need the URL of the existing spreadsheet.

```
id =
    "https://docs.google.com/spreadsheets/d/1J01M4ScGsAHb-bCUa0i575WWvQjXciQIV
    NOwVTrXDSe/edit#gid=2303744702"; // modify the URL
email = "yourgoogleaccount@gmail.com"; // modify the Google account
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Google Sheet Export(
        dt,
        Email( email ),
        Spreadsheet( id ),
        Sheet Name( "Example 1" )
);
```

"1" is returned in the log when the export is successful.

Note: See *Using JMP* for more information about security, country restrictions, and more.

Import SAS Data Sets

Open a SAS file as a data table without connecting to a SAS server.

```
sasxpt = Open( "$SAMPLE_IMPORT_DATA/carpoll.xpt" );
```

To convert the labels to column headings, include the `Use Labels for Var Names` argument.

```
sasdbf = Open( "$SAMPLE_IMPORT_DATA/Bigclass.sas7bdat", Use Labels for Var
    Names( 1 ));
```

.xpt and .stx file formats are also supported.

On Windows, you can also open SAS data sets from a SAS server. See "Connect to a SAS Metadata Server" on page 789 in the "Extending JMP" chapter.

Import Web Pages and Remote Files

You can import data from websites or from other computers. The data might be a JMP data table, a table defined in a web page, or another file type that JMP supports.

Open or Import Data from a Website

In the `Open()` command, specify the quoted URL to open a file from a website. You can open JMP data tables or other supported file types this way.

```
Open( "http://company1.com/Repairs.jmp" );
Open( "http://company1.com/My Data.txt", text); // specify text on macOS
```

Import a Web Page

A web page can include data in tabular format. Import the table as a JMP data table:

```
Open( "http://company1.com", HTML Table( n ) );
```

n identifies which table you want to import. For example, to import the fourth table on the page, specify HTML Table(4). If you omit the value, only the first table on the page is imported.

JMP attempts to preserve the table header defined in a `<th>` HTML tag. The table header is converted to column headings in the data table. If the `<th>` tag is wrong or missing, use ColumnNames(*n*) to specify the *n*th row. By default, DataStarts(*n*) will be the next row, or you can specify the DataStarts row.

Import Images from a Web Page

You can import images on the web into a data table, using the Expression data type. The following example creates a new data table and adds two images to it.

```
dt = New Table( "test", New Column( "Image", Expression ) );
dt << Add Rows( 1 );
dt:Image = New Image( Open(
    "https://www.jmp.com/support/help/en/16.0/jmp/images/iris_first.gif" )
    );
dt << Add Rows( 1 );
dt:Image = New Image( Open(
    "https://www.jmp.com/support/help/en/16.0/jmp/images/trimeth.png" ) );
```

Import Images from a Directory

You can use Multiple File Import() and an Event Handler script to create a data table of thumbnail images that link to a directory of full-size pictures. This example sets the cell height so that it fits the height of the tallest thumbnail.

Note: Typically, the Multiple File Import option is for importing multiple text files. However, this example shows how to use the option to import images.

```
path = "$SAMPLE_IMAGES/"; // images are in this directory

{dtx} = Multiple File Import(
    <<Set Folder( path ),
    <<Set Name Filter( "*.jpg; " ),
    <<Set Name Enable( 1 ), /* enable the ability to filter by name,
    in this example, files with the .jpg extension */
    <<Set Add File Name Column( 1 ), // must be included but can be hidden
    <<Set Add File Size Column( 1 ) // for Show Sizes at end
```

```
) << Import Data;

/* FileName is the column that contains the image file name.
The column is required for the image links to work, but you don't need
to see it. */
dtx:FileName << Hide;
maxheight = 0;

For Each Row( // recreate each image to minimize size of data table
    x = Eval( dtx:Picture );
    x << Scale( .1 ); // scale the image by 10%
    m = x << Get Pixels; // compress the image by recreating it
    dtx:Picture = New Image( m );
    maxheight = Max( maxheight, (dtx:Picture[] << Get Size)[2] );
    // width x height is returned, [2] gets the height
);

Eval( /* insert the value in the path variable into the Click handler. The
    path variable won't be available later, but the path value is needed to
    concatenate with the file name column's value to build a link to the image
    on the computer. */
    Eval Expr(
        dtx:picture << Set Property(
            "Event Handler",
            Event Handler( // Event Handler column property
                Click( JSL Quote( Function( {thisTable, thisColumn, iRow},
                // insert links to images
                Open( Expr( path )||Char( thisTable:FileName[ iRow ] ) ); ); ) ),

                Tip( JSL Quote( Function( {thisTable, thisColumn, iRow}, "Open " ||
    Char( thisTable:FileName[ iRow ] ) || " in your viewer."; ); ) ),

                // color of the link -- blue
                Color( JSL Quote( Function( {thisTable, thisColumn, iRow}, 5; );) )
            )
        )
    )
);

// add two pixels to the maximum height
dtx << Set Cell Height( maxheight + 2 );

/* save the file to show that the data table is much smaller than the
original file */
dtx << Save( "$TEMP/Sample Images.jmp" );
Show( File Size( "$TEMP/Sample Images.jmp" ), Col Sum( dtx:FileSize ) );
```

Import a File from a Shared Computer

JMP can import files stored on a shared computer, such as another computer or a network drive. The following example shows how to open a file from Windows:

```
Open( "\\Data\Repairs.jmp" ); // UNC name
Open( "z:/Data/Repairs.jmp" ); // mapped letter drive
Open( "Repairs.jmp" ); // relative path in which the script is in the same
    directory as the file that you are importing
```

macOS doesn't have a concept like UNC naming or drive letters. Instead, you need to mount the volume through the user interface. It is possible to do it from a script, but you would need to have connected interactively at least once and stored the password in your keychain.

Here is an example that connects to a remote volume from either macOS or Windows:

```
mount = Function( {server},
    {Default Local},

    If( Host is( Mac ),
        Run Program(
            Executable( "/usr/bin/osascript" ),
            // replace ^server^ with your server name throughout the script
            Options( {"-e", Eval Insert( "mount volume \!"smb://^server^\!"" )} ),
            Read Function( "text" )
        );
        server = Concat Items( Remove( Words( server, "/" ), 1 ), "/" );
        Eval Insert( "/Volumes/^server^" );
    ,
        Eval Insert( "\!\\!\^server^" )
    )
);


users = mount( "data.company.com/Users" );
Files In Directory( Eval Insert( "^users^/Smith" ) );
```

Import JSON Data

JSON (JavaScript Object Notation) is a data interchange format that is based on JavaScript. It shares many of the same principles as languages based on C and is easy to read and write. Because of these characteristics, JSON is highly portable and can be imported into JMP.

Table 9.1 Representations of JSON in JSL

JSON	JSL
array	list (sequence of values)
object	associative array (set of named values)
number	number
string	string
true	1.0
false	0.0
null	. (missing value)

JSON consists of name-value pairs in an associative array. For example, here is JSON code that shows name-value pairs:

```
"[ { \!"key1\!": \!"a\!", \!"key2\!": \!"b\!"}, {\!"key1\!": \!"c\!",
   \!"key2\!": \!"d\!" }, {\!"key2\!": \!"f\!", \!"key1\!":\!"e\!",
   \!"key3\!": \!"g\!"}]";
```

Each list is imported as a column in the data table. "key1" is the name of the first column. The first value in the column is "a". The second value is "c", and the third value is "e".

Use `JSON To Data Table()` to convert the JSON into a data table.

```
js = "[ { \!"key1\!": \!"a\!", \!"key2\!": \!"b\!"}, {\!"key1\!": \!"c\!",
   \!"key2\!": \!"d\!" }, {\!"key2\!": \!"f\!", \!"key1\!":\!"e\!",
   \!"key3\!": \!"g\!"}]";
d = JSON To Data Table( js ); // convert JSON to a data table
d << New Data View; // make a view of the data table
```

Figure 9.6 Imported JSON Data

You can also use `JSON To List()` to convert a string containing JSON into a list.

```
list = JSON To List(
```

```
    "[ { \!"name\!": \!"KATIE\!", \!"age\!": 12, \!"sex\!": \!"F\!",
    \!"height\!": 59, \!"weight\!": 95 }, { \!"name\!": \!"LOUISE\!",
    \!"age\!": 12, \!"sex\!": \!"F\!", \!"height\!": 61, \!"weight\!": 123 }, {
    \!"name\!": \!"JANE\!", \!"age\!": 12, \!"sex\!": \!"F\!", \!"height\!":
    55, \!"weight\!": 74 } ]"
);
Show( list );
    list = {{{"name", "KATIE"}, {"age", 12}, {"sex", "F"}, {"height", 59},
        {"weight", 95}}, ...}}};
```

Exporting JSL to JSON

As JSON Expr() converts an associative array to a JSON string. You can then export the data
into a JSON document.

```
list_of_associative_arrays = {["key1" => "a", "key2" => "b"], ["key1" => "c",
"key2" => "d"], ["key2" => "f", "key1" => "e", "key3" => "g"]};
jx = As JSON Expr( list_of_associative_arrays );
    "[{\!"key1\!":\!"a\!",\!"key2\!":\!"b\!"}..."[{\!"key1\!":\!"a\!",\!"key2\
        !":
    \!"b\!"}
```

Note: To import the data table as invisible, include the argument Invisible(1) in JSON To
Data Table() and omit New Data View.

Import HDF5 Files

Hierarchical Data Format, Version 5 (HDF5) is a portable file format for storing data. An
HDF5 file consists of groups and datasets. When you import the file, JMP opens a group to
present the names of the inner datasets.

JMP handles only tables with numeric (integer, float, double) and string types, and compound
files with three or fewer dimensions that contain only simple types.

You can import up to 1,000,000 columns and an unbounded number of rows.

Here is the syntax:

```
Open( "filename.h5", {"list_of", "dataset_names"});
```

Errors are written to the log when invalid data set names are passed.

Import ESRI Shapefiles

An ESRI shapefile is a geospatial vector data format used to create maps. JMP imports
shapefiles as data tables. A .shp shapefile consists of coordinates for each shape. A .dbf
shapefile includes values that refer to regions. To create maps in JMP, you modify the structure
of the data and save the files with specific suffixes.

The following example imports a .shp file and saves it with the -XY suffix.

```
dt = Open( "$SAMPLE_IMPORT_DATA/Parishes.shp"),
:X << Format( "Longitude DDD", 14, 4 );
:Y << Format( "Latitude DDD", 14, 4 ) );
dt << Save( "c:/Parishes-XY.jmp" );
```

Save the .dbf file with the -Name suffix.

```
dt = Open( "$SAMPLE_IMPORT_DATA/Parishes.dbf" );
dt << Save( "c:/Parishes-Name.jmp" );
```

Restructuring the data requires several steps, including adding a Map Role column property to names in the -Name.jmp file. See *Essential Graphing*.

Import a Database

`Open Database()` opens a database using Open Database Connectivity (ODBC) and extracts data into a JMP data table. See the "Database Access" on page 783 in the "Extending JMP" chapter.

JMP also converts DataBase Files (.dbf) files to data table format.

```
sasdbf = Open( "$SAMPLE_IMPORT_DATA/Bigclass.dbf",
    Use Labels for Var Names( 1 )
);
```

Set the Current Data Table

Tip: Be careful about assuming that the data table that you want is the current data table, even if you set it to be earlier in your script, because interim actions could change that.

A data table becomes the current data table in the following instances:

• When you open it like this:

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

• When you create a new table like this:

```
dt2 = New Table( "Cities" );
```

To switch to another open data table, specify the name of the data table in `Current Data Table()`:

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = New Table( "Cities" );
Current Data Table( dt1 ); // make Big Class.jmp the current data table
```

`Current Data Table()` can also take a scriptable object reference as an argument. The following expression would make the data table in use by the second Bivariate object the current data table:

```
Current Data Table( Bivariate[2] );
```

For more information about using references to analysis platform objects, see "Send Messages to a Platform" on page 475 in the "Scripting Platforms" chapter.

Name a Data Table

Assign a name to a data table by sending it the `Set Name` message. The argument is a filename in quotation marks, or something that evaluates to a filename.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Set Name( "New Big Class.jmp" );

s = "New Big Class";
dt << Set Name( s );
```

To retrieve the name, send a `Get Name` message to the data table:

```
dt << Get Name();
    "New Big Class"
```

Save a Data Table

To save a data table, send a `Save` message to the data table. Here are some examples:

```
dt << Save(); // save using the current name
dt << Save( "Newest Big Class.jmp" ) // save as a new file
dt << Save( "c:/My Data/New Big Class.jmp" );  // save as a new file
dt << Save( "My Table", JMP( 5 ) ); // save as JMP 5 table
dt << Save("") // prompt to select the directory and save in the desired
   format
dt << Save( "Big Class.xlsx" ); // save as a Microsoft Excel file
Create Excel Workbook( "c:/MyData/data.xlsx", {"Abrasion", "Big Class"},
{"Abrasive", "Class"} ); // save several data table as a workbook
```

Note: If you specify the filename with no path and have not set the default directory, the file is saved on your primary partition (Windows) or in your *<username>*/Documents folder (macOS). For more information about setting the default directory, see "Relative Paths" on page 149 in the "Types of Data" chapter.

On Windows, saving with a `.txt` extension exports according to the Text Export preferences. On macOS, add `Text` as a second argument to the `Save` function:

```
dt << Save( "New Big Class.txt", Text );
```

If you plan to set the name of a data table and later send the Save message, you can just specify the name in a Save message.

```
dt << Set Name( "New Big Class.jmp" );
dt << Save();
```

works the same as

```
dt << Save( "New Big Class.jmp" );
```

Including Save and the pathname is also an alternative to using Save As along with the pathname.

Revert to a Saved Data Table

To return to the most recently saved data table, send a Revert message to the data table.

```
dt << Revert();
```

Hide a Data Table

There are two ways to hide a data table if the user does not need to see the table: by opening it as invisible or private.

Invisible Data Tables

An *invisible* data table is hidden from view but linked to analyses that you run on it. Open a data table as an invisible file:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", "invisible" );
```

Here is an example of creating an invisible table, called Abc, that has ten rows. One column is named X.

```
dt = New Table( "Abc", "invisible", New Column( "X" ), Add Rows( 10 ) );
```

To find out if an open data table is invisible, pass the Boolean Has Data View message to the data table object. The following expression returns 0 (false) if the data table is invisible:

```
dt << Has Data View();
```

After you are finished with an invisible data table, do not forget to close it with the Close() function. Otherwise, the data table remains in memory until you quit JMP.

Showing an Invisible Data Table

The Show Window() function shows an invisible data table.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", "invisible" );
```

```
dt << Show Window( 1 );
```

The user has the option of opening the table from the JMP interface. On Windows, an invisible data table appears in the Window List on the Home Window and the **Window > Unhide** menu. On macOS, the data table appears in the JMP Home window and the **Window > Hidden** menu.

Private Data Tables

Completely hide the data table from view by including the private argument:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", "private" );
```

A private data table enables you to avoid exposing the table to general use. Sometimes a script creates a private data table to store private data generated by a process. It is not for saving memory.

See "Private, Invisible, and Visible" on page 330.

Print a Data Table

Print a data table by sending it a Print Window message. JMP uses your computer's default printer settings.

```
dt << Print Window();
```

This message also applies to other display boxes.

Resize a Data Table

dt << Optimize Display forces the data table to remeasure all of its columns and zoom to the best-sized window.

```
dt << Optimize Display;
```

Close a Data Table

To close a data table, use a Close command with the data table's reference as the argument. Data tables with unsaved changes are saved automatically, including any linked tables. Reports and graphs generated from the data tables are also closed.

To save and close the data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Close( dt );
```

To close the data table without saving changes:

```
Close( dt, No Save );
```

To save a copy with a new name and then close the original version:

```
Close( dt, Save("c:/My File.jmp" ) );
```

You can also close all data tables at once and either save or discard changes:

```
Close All( Data Tables );
Close All( Data Tables, No Save );
```

To close all private or invisible data tables, specify the appropriate argument:

```
Close All( Data Tables, private );
Close All( Data Tables, invisible );
```

Tip: Close All() also works with journals. Specify "Journals" in the argument.

Set and Get a Data Table

To specify the data table in a display box, for example, in New Data Box(), use Set Data Table() and Get Data Table() to set and return a data table.

The following example uses the Big Class.jmp and Cities.jmp sample data tables. The sample data tables are opened invisibly. In a new window, a new data box is created containing Big Class.jmp. After waiting 1 second, the data box contents change to display cities.jmp.

```
dtC = Open( "$SAMPLE_DATA/Cities.jmp", "invisible" );
dtA = Open( "$SAMPLE_DATA/Big Class.jmp", "invisible" );
nw = New Window( "My Table", H List Box( dtBox = dtA << New Data Box() ) );
Wait( 1.0 );
dtBox << Set Data Table( Data Table( "Cities.jmp" ) );
```

Perform Actions on All Open Data Tables

If you want to perform an action on all of the data tables that are currently open (except private tables), use N Table() to get a list of references to each one:

```
dt1 = Open( "$SAMPLE_DATA/Cities.jmp" );
dt2 = Open( "$SAMPLE_DATA/Big Class.jmp" );
openDTs = List();
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) );
);
```

openDTs now is a list of references to all open data tables. You can send messages to any one by using openDTs(n). You can use a for loop to send messages to all of the open data tables. This loop adds a new column named My Column to each open data table.

```
For( i = 1, i <= N Items( openDTs ), i++,
```

```
      openDTs[i] << New Column( "My Column" );
   );
```

If you just want a list of table names and not references, use the `Get Name` message:

```
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) << Get Name());
   );
```

You can then use the list of table names to put the names of all of the open data tables in a list box (window) so that the user can choose and perform tables operations. Or, you can write the names out to a file.

Create Journals

Journals consist of JMP graphs and reports, graphics, text, and links to items such as web pages and files. They make it easy to reuse content in presentations and import into other documents.

Note: `Layout` is deprecated and will be removed in a future release. Use `Journal` instead.

Here is the basic syntax for journals:

```
dt << Journal;
```

You can also create a new journal window and immediately add to it within the `New Window()` command. The following examples create an empty untitled journal:

```
New Window( <<  Journal);
```

The following examples create an empty journal named "Sales":

```
New Window( "Sales", << Journal );
```

Here is an example of creating a journal named "Test Buttons" that contains two buttons.

```
New Window( "Test Buttons", << Journal,
   Button Box( "Test One", New Window( "Hi there1", << Modal ) ),
   Button Box( "Test Two", New Window( "Hi there2", << Modal ) ) );
```

Capturing Journals

There is an optional argument in the journal and journal window commands. This argument captures (or freezes) the current display. That is, it converts the display tree with just a picture.

Here are the four options for arguments:

• Makes a bitmap "snapshot" of the display when it is sent to the journal, rather than sending a clone of the more editable display box structure to the journal. By freezing the display into a bitmap, references to variables from scripts in graphs are less problematic.

```
<< Journal("Freeze All");
```

- Similar to `Freeze All` but only within areas of a report called pictures.

```
<< Journal("Freeze Pictures");
```

- Similar to `Freeze Pictures` but only within frame boxes (which are within pictures).

```
<< Journal("Freeze Frames");
```

- Similar to `Freeze Frames` but only for the frame boxes that have scripts to draw something on them.

```
<< Journal("Freeze Frames with Scripts");
```

Current Journal

`Current Journal()` returns a reference to the display box at the top of the current journal display window. If no journal is open, one is created. There are no arguments.

You can add to the journal using the Append command. The following example adds a text box to the bottom of the current journal:

```
Current Journal() <<  Append( Text Box( "Hello World" ) );
```

You can also find items in an existing journal by enclosing the search specification in square brackets. Suppose that the current journal includes the string "Parameter Estimates". The following script appends a text box to the bottom of that journal:

```
Current Journal()["Parameter Estimates"] << Append( Text Box( "Asterisks show
    items significant at 0.05" ) );
```

Create Excel Workbooks

You can save open data tables in a new or existing Excel workbook by using the `Create Excel Workbook()` function. The first example adds all open data tables without changing the worksheet names:

```
dt1 = Open( "$SAMPLE_DATA/Big Class Families.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
Create Excel Workbook( "c:/MyWorkbook1.xlsx" );
```

The following example creates MyWorkbook2.xlsx by combining the open Big Class Families.jmp and San Francisco Crime.jmp sample data tables.

```
dt1 = Open( "$SAMPLE_DATA/Big Class Families.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );

// specify the data tables
tableList = {"Big Class Families", "San Francisco Crime"};
```

```
// specify the worksheets (optional)
sheetList = {"Families", "Crime"};
Create Excel Workbook( "c:/MyWorkbook2.xlsx", tableList, sheetList );
```

Use data table references to create the workbook.

```
dt1 = Open( "$SAMPLE_DATA/Big Class Families.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
Create Excel Workbook(
   "c:/MyWorkbook3.xlsx",
   {dt1, dt2}, // data table references
   {"Families", "Crime"}
);
```

Create Excel Workbook() also accepts a mix of data table references and data table names in the list. For example, {dt1, dt2} above can also be expressed as {dt1, "San Francisco Crime"}.

Notes:

- The saved file is an .xlsx file; the .xls format is not supported.

- The JMP data tables must conform to the maximum Excel limit of 1 million rows and 16,535 columns, or the data will be truncated in Excel.

- Data table references or data table names must always be in a list, even if the list has only one item.

Prevent Changes to a Data Table

To prevent editing cells in the data table, use Set Edit Lock with the Modify Cells argument.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Set Edit Lock( "Modify Cells" );
```

To prevent adding or deleting columns and rows:

```
dt << Set Edit Lock( "Add Columns", "Delete Dolumns");
dt << Set Edit Lock( "Add Rows", "Delete Rows");
```

To get a list of data table operations that are locked, use Get Edit Lock:

```
dt << Get Edit Lock();
   {"Modify Cells", "Add Rows", "Delete Rows", "Add Columns", "Delete Columns"}
```

To allow a specific data table operation again, use Clear Edit Lock:

```
dt << Clear Edit Lock( "Modify Cells" );
```

You can lock all three data table components (cells, columns, and rows) at once by using Lock Data Table.

```
dt << Lock Data Table( 1 );
```

Advanced Data Table Scripting

This section covers more advanced actions that you can perform on a data table.

- "Select Columns in a Data Table"
- "Store Summary Statistics in Global Variables"
- "Create a Table of Summary Statistics"
- "Subset a Data Table"
- "Add a Data Filter"
- "Sort a Data Table"
- "Stack Values in a Data Table"
- "Split Values in a Stacked Data Table"
- "Transpose a Data Table"
- "Vertically Concatenate Data Tables"
- "Horizontally Concatenate Data Tables"
- "Virtually Join Data Tables"
- "Replace Data in Data Tables"
- "Create a Table Using Tabulate"
- "Copy Data Tables"
- "Find Missing Data Patterns"
- "Compare Data Tables"
- "Create a Summary Table"
- "Subscribe to a Data Table"
- "Move Data Between Matrices and Data Tables"
- "Add Hyperlinks to Text Columns"

Select Columns in a Data Table

To select one or more columns, send the Select Columns message to the data table.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
clist = {:Height, :Weight};
dt << Select Columns( clist );
```

To select all columns, send the Select Columns message with the All argument to the data table.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Columns( all );
```

Store Summary Statistics in Global Variables

The `Summarize` command collects summary statistics for a data table and stores them in global variables. The `Summarize` command is different from the `Summary` command, which also calculates summary statistics, but presents them in a new data table.

The first argument is an optional data table reference. Include it if more than one data table might be open.

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/Animals.jmp" );
Summarize( dt1,
    exg = By( :sex ),
    exm = Mean( :height )
);
Show( exg );
Show( exm );
```

Named arguments include the following: `Count`, `Sum`, `Mean`, `Min`, `Max`, `StdDev`, `First`, `Corr`, and `Quantile`. These statistics can be calculated only for numeric columns. Each argument takes a data column argument.

Notes:

- If a *name*=By(*groupvar*) statement is included, then a list of subgroup statistics is assigned to each *name*.

- `Count` does not require a column argument, but it is often useful to specify a column to count the number of nonmissing values.

- `Quantile` also takes a second argument for specifying which quantile, such as 0.1 for the 10th percentile.

- When you summarize data that contain row states, the summary statistics are not calculated. An error appears in the log instead.

Note: Excluded rows are excluded from `Summarize` calculations. If all data are excluded, Summarize returns lists of missing values. If all data have been deleted (there are no rows), Summarize returns empty lists.

The following example uses the Big Class sample data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
    a = By( :age ),
    c = Count,
```

```
        sumHt = Sum( :height ),
        meanHt = Mean( :height ),
        minHt = Min( :height ),
        maxHt = Max( :height ),
        sdHt = Std Dev( :height ),
        q10Ht = Quantile( :height, .10 )
    );
    Show( a, c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht );
```

Because the script included a By group, the results are a list and six matrices:

```
    a = {"12", "13", "14", "15", "16", "17"}
    c = [8, 7, 12, 7, 3, 3]
    sumHt = [465, 422, 770, 452, 193, 200]
    meanHt = [58.125, 60.28571428571428, 64.16666666666667, 64.57142857142857,
        64.33333333333333, 66.66666666666667]
    minHt = [51, 56, 61, 62, 60, 62]
    maxHt = [66, 65, 69, 67, 68, 70]
    sdHt = [5.083235752381126, 3.039423504234876, 2.367712103711172,
        1.988059594776032, 4.041451884327343, 4.163331998932229]
    q10Ht = [51, 56, 61.3, 62, 60, 62]
```

You can format the results using Table Box.

```
    New Window( "Summary Results",
        Table Box(
            String Col Box( "Age", a ),
            Number Col Box( "Count", c ),
            Number Col Box( "Sum", sumHt ),
            Number Col Box( "Mean", meanHt ),
            Number Col Box( "Min", minHt ),
            Number Col Box( "Max", maxHt ),
            Number Col Box( "SD", sdHt ),
            Number Col Box( "Q10", q10Ht )
        )
    );
```

Figure 9.7 Results from Summarize

Age	Count	Sum	Mean	Min	Max	SD	Q10
12	8	465	58.125	51	66	5.08324	51
13	7	422	60.2857	56	65	3.03942	56
14	12	770	64.1667	61	69	2.36771	61.3
15	7	452	64.5714	62	67	1.98806	62
16	3	193	64.3333	60	68	4.04145	60
17	3	200	66.6667	62	70	4.16333	62.

You can add totals to the window:

```
    dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

    Summarize(
```

```
    a = By( :age ),
    c = Count,
    sumHt = Sum( :height ),
    meanHt = Mean( :height ),
    minHt = Min( :height ),
    maxHt = Max( :height ),
    sdHt = Std Dev( :height ),
    q10Ht = Quantile( :height, .10 )
);

Summarize(
    tc = Count,
    tsumHt = Sum( :height ),
    tmeanHt = Mean( :height ),
    tminHt = Min( :height ),
    tmaxHt = Max( :height ),
    tsdHt = Std Dev( :height ),
    tq10Ht = Quantile( :height, .10 )
);

Insert Into( a, "Total" );
c = c |/ tc;
sumHt = sumHt |/ tsumHt;
meanHt = meanHt |/ tmeanHt;
minHt = minHt |/ tminHt;
maxHt = maxHt |/ tmaxHt;
sdHt = sdHt |/ tsdHt;
q10Ht = q10Ht |/ tq10Ht;

New Window( "Summary Results",
    Table Box(
        String Col Box( "Age", a ),
        Number Col Box( "Count", c ),
        Number Col Box( "Sum", sumHt ),
        Number Col Box( "Mean", meanHt ),
        Number Col Box( "Min", minHt ),
        Number Col Box( "Max", maxHt ),
        Number Col Box( "SD", sdHt ),
        Number Col Box( "Q10", q10Ht )
    )
);
```

Figure 9.8 Summarize with Total

Age	Count	Sum	Mean	Min	Max	SD	Q10
12	8	465	58.125	51	66	5.08324	51
13	7	422	60.2857	56	65	3.03942	56
14	12	770	64.1667	61	69	2.36771	61.3
15	7	452	64.5714	62	67	1.98806	62
16	3	193	64.3333	60	68	4.04145	60
17	3	200	66.6667	62	70	4.16333	62
Total	40	2502	62.55	51	70	4.24234	56.2:

If you do *not* specify a By group, the result in each name is a single value:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
// a = By( :age ),
   c = Count,
   sumHt = Sum( :height ),
   meanHt = Mean( :height ),
   minHt = Min( :height ),
   maxHt = Max( :height ),
   sdHt = Std Dev( :height ),
   q10Ht = Quantile( :height, .10 )
);
Show( c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht );
   c = 40;
   sumHt = 2502;
   meanHt = 62.55;
   minHt = 51;
   maxHt = 70;
   sdHt = 4.24233849397192;
   q10Ht = 56.2;
```

Summarize supports multiple By groups. For example, in Big Class.jmp, proceed as follows:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize( g = By( :age, :sex ), c = Count() );
Show( g, c );
   g = {{"12", "12", "13", "13", "14", "14", "15", "15", "16", "16", "17", "17"},
       {"F", "M", "F", "M", "F", "M", "F", "M", "F", "M", "F", "M"}}
   c = [5,3,3,4,5,7,2,5,2,1,1,2]
```

If you specify a By group, the results are always matrices. Otherwise, the results are scalars.

Create a Table of Summary Statistics

The Summary command creates a new table of summary statistics according to the grouping columns that you specify. Do not confuse Summary with Summarize, which collects summary statistics for a data table and stores them in global variables. See "Store Summary Statistics in Global Variables" on page 364.

```
summDt = dt << Summary(
```

```
      Group( groupingColumns ),
      Subgroup( subGroupColumn ),
      Statistic( columns ),// where statistic is Mean, Min, Max, Std Dev, and so
      on.
      Output Table Name( newName ) );
```

The following example creates a new table with columns for the mean of height and weight by age, and the maximum height and minimum weight by age:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
summDt = dt << Summary(
   Group( :age ),
   Mean( :height, :weight ), Max( :height ), Min( :weight ),
   Output Table Name( "Height-Weight Chart" ) );
```

Tip: Output Table Name can take a quoted string or a variable that is a string.

By default, a summary table is linked to the original data table. If you want to produce a summary that is not linked to the original data table, add this option to your Summary message:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
summDt = dt << Summary( Group( :age ), Mean( :height ),
   Link to Original Data Table( 0 )
);
```

You can choose to add marginal statistics for grouping variables to the output columns of your data table. JMP also adds a row to the end of the data table that summarizes each level of the grouping variables. To add marginal statistics, add this option to your Summary message:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
summDt = dt << Summary( Group( :age ), Mean( :height ),
   Include marginal statistics
);
```

You can specify the format of the statistics column name using statistics column name format(). Table 9.2 shows the option and examples:

Table 9.2 Statistics Column Name Format Options and Examples

Option	Example
stat (column)	Mean (Profits ($M))
column	Profits ($M)
stat of column	Mean of Profits ($M)
column stat	Profits ($M) Mean

For example, add an option like this to your Summary message:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
summDt = dt << Summary( Group( :age ), Mean( :height ),
    statistics column name format( "stat of column" )
);
```

The height column is named Mean of height.

Subset a Data Table

Subset() creates a new data table from rows that you specify. If you specify no rows, Subset uses the selected rows. If no rows are selected or specified, it uses all rows. If no columns are selected or specified, it uses all columns. And if Subset has no arguments, the Subset window appears.

```
dt << Subset((<"Private">|<"Invisible">), <Selected Columns>, <Columns(column
    list)>, <Selected Rows>, <Rows([number, number, ...])>, <By(column
    list)>, <Sampling Rate(fraction)>, <Sample Size(integer)>,
    <Stratify(column list)>, <Link to Original Data Table(Boolean)>, <Copy
    Formula(Boolean)>, <Suppress Formula Evaluation(Boolean)>, <Keep by
    Column>);
```

Note: For more arguments, see the Scripting Index in the Help menu.

For example, using Big Class.jmp, to select the columns for all rows in which the age is 12:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( Selected( Row State() ) = (:age == 12) );
subdt = dt << Subset( Output Table Name( "Subset" ) );
```

To select three columns and all rows, run this script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
subDt1 = dt << Subset(
    Columns( :name, :age, :height ),
    Output Table Name( "Big Class 2" )
);
```

To select specified rows of two columns and link the two data tables, run this script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
subDt2 = dt << Subset(
    Columns( :name, :weight ),
    Rows( [2, 4, 6, 8] ),
    Linked
);
```

To select the columns for all rows in which the age is 12, run this script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age == 12 );
dt << Subset( ( Selected Rows ), Output Table Name( "Subset" ) );
```

To verify that rows were selected and then conditionally proceed to subset the current data table or open another data table, run this script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age == 12 );

/* If the condition selected no rows, open Fitness.jmp.
   Else, generate the desired subset table of Big Class.jmp.*/
If( N Rows( dt << Get Selected Rows ) == 0,

newdt = Open( "$SAMPLE_DATA/Fitness.jmp" ),
   newdt = dt << Subset( All Columns( 1 ), Selected Rows( 1 ) )
);
```

Add a Data Filter

The Data Filter provides a variety of ways to identify subsets of data. Using Data Filter commands and options, you can select complex subsets of data, hide these subsets in plots, or exclude them from analyses.

Here is the basic syntax:

```
dt << Data Filter(<Columns>, <Mode(...)>, <Add Filter(...)>, <Show Columns(),
   <Selector>, <Animation>, <Animate>, <Clear>, <Delete All>, <Delete>,
   <Filter Columns>, <Filter Column>, <Filter Group>, <Add Filter>, <Add>,
   <Match>, <Mode>, <Display>, <Where>, <Make Subset>, <Display>, <Show
   Histograms and Bars>)
```

You can send an empty Data Filter message to a data table, and the initial Data Filter window appears, showing the Add Filter Columns panel that lists all the variables in the data table.

Mode takes three arguments, all of which are optional: Select(*Boolean*) Show(*Boolean*), Include(*Boolean*). These arguments turn on or off the corresponding options. The default value for Select is true (1). The default value for Show and Include is false (0).

Add Filter() adds rows and builds the WHERE clauses that describe a subset of the data table. Here is the basic syntax:

```
Add Filter( Columns( col, ... ), Where( ... ), ... )
```

To add columns to the data filter, list the columns names separated by commas. Note that this is not a list data structure.

You can define one or more WHERE clauses to specify the filtered columns:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
df = dt << Data Filter(
    Mode( Show( 1 ) ),
    Add Filter(
        Columns( :age, :sex, :height ),
        Where( :age == {13, 14, 15} ),
        Where( :sex == "M" ),
        Where( :height >= 52 & :height <= 65 )
    )
);
```

This script selects 13, 14, and 15-year-old males with a height greater than or equal to 52 and less than or equal to 65. In this filtered data, you can instead select the excluded ages by adding the Invert Selection message to the preceding script.

```
df << ( Filter Column( :age ) << Invert Selection );
```

In this example, ages other than 13, 14, and 15 in the filtered data are selected.

You can also use Add Filter() to select matching strings from columns with the Multiple Response property or Multiple Response modeling type.

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
df = dt << Data Filter(
    Location( {437, 194} ),
    Add Filter(
        Columns( :Brush Delimited ),
        Match None( Where( :Brush Delimited == {"Before Sleep", "Wake"} ) ),

        /* Define the display options. "Brush Delimited" appears above
        the list of values. Size is the size of the filter list. The
        values appear next to check boxes. */
        Display( :Brush Delimited, Size( 121, 70 ), Check Box Display )
        )
);
```

This script selects rows with values in the Brush Delimited column that do not match either of the specified values ("Before Sleep", "Wake"). Other available scripting options include Match Any, Match All, Match Exactly, and Match Only. See *Using JMP* for more information about the Multiple Response property and the Multiple Response modeling type.

You can also send messages to an existing Data Filter object:

```
Clear(), Display( ... ), Animate(), Mode(), ...
```

Clear takes no arguments and clears the data filter.

To prevent the data filter from appearing when the script is run, set Show Window to 0:

```
obj  = ( dt << Data Filter( Show Window( 0 ) ) );
```

To specify which values to exclude, use the != operator. The following example excludes ages 16 and 17 from the filtered values.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
df = dt << Data Filter(
    Add Filter( Columns( :age, :sex ) ),
    Match( Columns( :age, :sex ),
        Where( :sex = "M" ), Where( :age != {16, 17} )
    )
);
```

To omit the Data Filter outline and red triangle menu, specify No Outline Box.

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
obj = dt << Data Filter(
    Add Filter( Columns( :Region, :POP ), Where( :Region == {"C", "N"} ) ),
    Mode( Select( 0 ), Show( 0 ), Include( 1 ) ),
    No Outline Box( 1 )
);
```

Add a Local Data Filter

A local data filter enables you to filter data in an individual report window without affecting other reports. When the local data filter is added to the platform, the reference to the data filter is returned. This provides an easy way to get access to the filter.

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
biv = dt << Bivariate(
    Y( :x ),
    X( :y ),
    Automatic Recalc( 1 ),
    Fit Line( {Line Color( {213, 72, 87} )} )
);

dfObj = biv << Local Data Filter(
// include Show Histograms and Bars( 0 ) to hide them in the filter.
    Location( {566, 47} ),
    Mode( Select( 0 ), Include( 1 ) )
);

dfObj << Add Filter(
    Columns( :Region, :POP ),
    Where( :Region == {"C", "N"} )
);
```

Track Data Filter Changes

Make Filter Change Handler enables you to track changes to the data filter.

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dist = Distribution(
    Automatic Recalc( 1 ),
    Continuous Distribution( Column( :POP ) )
);
filter = dist << Local Data Filter(Add Filter( columns( :Region ) ));
f = Function( {a}, Print( a ) );
rs = filter << Make Filter Change Handler( f );
```

Define the Context of a Data Filter

A data filter lets you interactively select complex subsets of data, hide the subsets in plots, or exclude them from analyses. Your selections affect all analyses of the data table.

Another option is to filter data from specific platforms or display boxes. Create a local data filter inside the Data Filter Context Box() function. This defines the context as the current platform or display box rather than the data table.

The following example creates a local data filter for a Graph Box. See Figure 9.9 for the output.

```
New Window( "Marker Seg Example",
    Data Filter Context Box(
        V List Box(
            dt << Data Filter( Local ),
            g = Graph Box(
                Frame Size( 300, 240 ),
                X Scale( Min( xx ) - 5, Max( xx ) + 5 ),
                Y Scale( Min( yy ) - 5, Max( yy ) + 5 ),
                Marker Seg( xx, yy, Row States( dt, rows ) ) )
            )
        )
    )
);
```

Tip: The preceding script is part of a larger script that first builds the arrays required for a marker seg. To experiment with this script, open the Local Data Filter for Custom Graph.jsl sample script.

Figure 9.9 Local Data Filter and Graph

A context box can also contain several graphs with one local filter. The filtering then applies to both graphs. The following script creates two bubble plots and one data filter in one context box. See Figure 9.10 for the output.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Shared Local Filter",
    Data Filter Context Box(
        H List Box(
            dt << Data Filter( Local ),
            Platform(
                dt,
                Bubble Plot( X( :weight ), Y( :height ), Sizes( :age ) )
            ),
            Platform(
                dt,
                Bubble Plot( X( :weight ), Y( :age ), Sizes( :height ) )
            )
```

```
      )
    )
  );
```

Tip: To experiment with this script, open the Local Data Filter Shared.jsl sample script.

Figure 9.10 Local Filter with Two Bubble Plots

Data filters can be hierarchical. One script might generate a data filter and a local data filter. The outer data filter for the data table determines which data is available for the local data filter. To illustrate this point, the following script produces both types of data filters as shown in Figure 9.11.

Tip: The following script is part of a larger script that first builds the arrays required for a marker seg. See the Local Data Filter for Custom Graph.jsl sample script for code that builds those arrays.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
...
New Window( "Marker Seg Example",
  Data Filter Context Box(
    V List Box(
      dt << Data Filter( Local ),
      g = Graph Box(
        Frame Size( 300, 240 ),
        X Scale( Min( xx ) - 5, Max( xx ) + 5 ),
        Y Scale( Min( yy ) - 5, Max( yy ) + 5 ),
        Marker Seg( xx, yy, Row States( dt, rows ) )
      )
    )
  )
);
```

Heights greater than or equal to 55 and less than or equal to 65 are initially filtered. Then the user can work with the local data filter to filter columns on the graph.

Figure 9.11 Data Filter Hierarchy

Sort a Data Table

Sort() rearranges the rows of a table according to the values of one or more columns, either replacing the current table or creating a new table with the results. Specify ascending or descending sort for each By column.

```
dt << Sort(("<Private>"|"<Invisible>"), <Replace Table>, By(columns),
    Order(Descending | Ascending), <Output Table Name(name))>
```

The following example creates a new data table based on Big Class.jmp that sorts the data in descending order by age and by name:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
sortedDt = dt << Sort(
    By( :age, :name ),
    Order( Descending, Ascending ),
```

```
      Output Table Name( "Sorted age name" ) );
```

You can also use an associative array to sort values in a column. The associative array gives you a quick look at unique values in a column. See "Sort a Column's Values in Lexicographic Order" on page 245 in the "Data Structures" chapter.

Stack Values in a Data Table

Stack() combines values from several columns into one column.

```
dt << Stack(
   Columns ( columns ),  // the columns to stack together
   Source Label Column ( "name" ), // to identify source columns
   Stacked Data Column ( "name" ), // name for the new stacked column
   Number of Series( n ), // stacks selected columns into two or more columns
   Contiguous, // specifies that the series consists of adjacent columns
   Keep ( columns ), // the columns to keep in the data table
   Drop ( columns ), // the columns to drop in the data table
   Output Table ( "name" ), // the name for the new data table
   Columns( columns ) ), // specify which columns to include in the stacked
   table
   Copy Formula( Boolean ), // copy the formula into the stacked column
   Drop All Other Columns( Boolean ), // omit the non-stacked columns from the
   new table
   "non-stacked columns"n )( Keep( col1, ... ) // the non-stacked columns to
   keep
   "non-stacked columns"n )( Drop( col1, ... ) // the non-stacked columns to
   omit
   Suppress Formula Evaluation( Boolean ) // do not evaluate the formula
```

For example, the following script stacks the weight and height columns in Big Class.jmp:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
StackedDt = dt << Stack(
    Columns( :weight, :height ),
    Source Label Column( "ID" ),
    Stacked Data Column( "Y" ),
    "Non-stacked columns"n )( Keep( :age, :sex ) ),
    Output Table( "Stacked Table" )
);
```

The Columns(columns) argument can take a list of columns, or an expression that evaluates to a list.

You can also stack multiple sets of columns into multiple columns.

```
dt = Open( "$SAMPLE_DATA/Blood Pressure.jmp" );
dt << Stack(
```

```
Columns(
    :BP 8M,
    :BP 12M,
    :BP 6M,
    :BP 8W,
    :BP 12W,
    :BP 6W,
    :BP 8F,
    :BP 12F,
    :BP 6F
),
Stacked Data Column( "BP" ),
Source Label Column( "Day" ),
Number of Series( 3 ), // number of sets in each series
Contiguous, /* If you want to stack vertically. Otherwise, omit the
argument. */

);
```

Split Values in a Stacked Data Table

Split() breaks a stacked column into several columns.

```
dt << Split(
    Split( columns ),
    // the column to split (required)

    Split by( column ),
    // the column to split by (required)

    Group(column),
    // split data within groups

    <Private>|<Invisible>,
    // resulting table is private or invisible

    Remaining Columns( Keep All | Drop All | Select( columns ) ),
    /* specify what to do with the remaining columns in the resulting table
    (Keep All by default) */

    <Copy formula( 0|1 )>,
    /* include column formulas from the source table in the resulting table
    (default is 1, true) */

    <Suppress formula evaluation( 0|1 )>,
    // stop any copied formulas from being evaluated (default is 1, true)
```

```
    Sort by Column Property,
    /* Sort the order of the output columns by the sort column property that is
    defined for the Split by column. */

    Output Table( "name" ));
    // generate the output to the table name specified
```

The following example reverses the previous example for Stack(), returning essentially the original table, except that the height and weight columns now appear in alphabetic order:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Stack(
    Columns( :weight, :height ),
    Source Label Column( "ID" ),
    Stacked Data Column( "Y" ),
    "Non-stacked columns"n( Keep( :age, :sex ) ),
    Output Table( "Stacked Table" )
);
dt2 = Data Table( "Stacked Table" );
dt2 << Split(
/* Split adds the column or columns that you want to split into several
new columns. */
    Split( Y ),
/* Split By adds the column whose values you want to use as the new column
    names, and as the basis for splitting the column. */
    Split By( ID ),
    Output Table( "Split" ) );
```

When multiple rows are mapped to the same row, only one value is kept.

```
dtabc = New Table( "abc",
 New Column( "Column 1", Character, Set Values( {"a", "a", "b", "b", "c"} )),
 New Column( "Column 2", Character, Set Values( {"x", "y", "x", "y", ""} )),
 New Column( "Column 3", Set Values( [1, 2, 3, 4, 5] ))
);
/* Split adds Column 3. You want to split the values (1, 2, 3..) into several
    new cols.
Split By adds Column 2. You want to use the values (x, y) as the new column
    names. */
dtabc << Split( Split By( :Column 2 ), Split( :Column 3 ));
```

Transpose a Data Table

Transpose creates a new data table by flipping a data table on its side, interchanging rows for columns and columns for rows. If you specify no rows, Transpose() uses the selected rows. If no rows are selected, it uses all rows.

```
dt << Transpose(
  "private", "invisible",
  columns( columns ),
  Rows( row matrix ),
  By ( column ),
  Label column name( "name" ),
  Output Table( "name" )
);
dt << Transpose(
  Columns( columns ),
  Rows( row matrix ),
  Output Table Name( "name" )
);
```

The following example transposes the height and weight columns in the Big Class.jmp sample data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
tranDt = dt << Transpose( Columns( :height, :weight ),
  Output Table Name( "Transposed Columns" ) );
```

Note: The simple transpose command dt << Transpose brings up the Transpose window. If you do not want the window to appear, invoke the transpose as dt << Transpose(no option).

Vertically Concatenate Data Tables

Concatenate, also known as a vertical join, combines rows of several data tables top to bottom.

```
dt << Concatenate( DataTableReferences,...,Keep Formulas,
  Output Table Name( "name" ) );
```

For example, if you have subsetted tables for males and females, you can put them back together using Concatenate:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :sex == "M" );
m = dt << Subset( Output Table Name( "M" ) );
dt << Invert Row Selection;
f = dt << Subset( Output Table Name( "F" ) );
both = m << Concatenate( f, Output Table Name( "Both" ) );
```

Or, instead of creating a new table containing all the concatenated data, you can append all the data to the current data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :sex == "M" );
```

```
m = dt << Subset( Output Table Name( "M" ) );
dt << Invert Row Selection;
f = dt << Subset( Output Table Name( "F" ) );
both = dt << Concatenate( m, f, "Append to First Table" );
```

Horizontally Concatenate Data Tables

Join, also known as horizontal join or concatenate, combines data tables side to side.

```
dt << Join( // message to the first table
   With( dataTable ), // the secondary data table
   Select( columns ), /* select columns from the main table
                      to add to the output table */
   Select With( columns ), /* select columns from the secondary table to
                           add to the output table */
   // join type; alternatives are
   Cartesian join, By Row Number, By matching columns(col1=col2, ...)
   Merge Same Name Columns, // merge columns with the same name
   Copy Formula( 0 ), // on by default; 0 turns it off
   Suppress Formula Evaluation( 0 ), // on by default; 0 turns it off
   Match Flag( 1 ), /* add the Match Flag column from the joined data table
   when you are matching by column */
   Update, /* replace the data in the main table with the corresponding data
   from the secondary table */
   // options for each table:
   Drop Multiples( Boolean, Boolean ), // include all rows from the data table
   Include Non Matches( Boolean, Boolean ), // include non-matching rows
   Preserve Main Table Order(), // maintain the order of the original data
   Output Table Name( "name" )); // the resulting table
```

To try this, first break Big Class.jmp into two parts:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
part1 = dt << Subset(
   Columns( :name, :age, :height ),
   Output Table Name( "Big Class 1" )
);
part2 = dt << Subset(
   Columns( :name, :sex, :weight ),
   Output Table Name( "Big Class 2" )
);
```

To make it a realistic experiment, rearrange the rows in part 2:

```
sortedPart2 = part2 << Sort( By( :name ), Output Table Name( "Sorted_Big
   Class" ) );
```

Now you have a data set in two separate chunks, and the rows are not in the same order, but you can join them together by matching on the column that the two chunks have in common.

```
part1 << Join(
    With( sortedPart2 ),
    By Matching Columns( :name == :name ),
    Preserve Main Table Order();
    Output Table Name( "Joined Parts" );
);
```

The resulting table has two copies of the name variable, one from each part, and you can inspect these to see how `Join` worked. Notice that you now have four Robert rows, because each part had two Robert rows (there were two Roberts in the original table) and `Join` formed all possible combinations.

Tip: To maintain the order of the original data table in the joined table (instead of sorting by the matching columns), include `Preserve Main Table Order()`. This function speeds up the joining process.

See the *JSL Syntax Reference* for more information about the Join arguments.

Virtually Join Data Tables

Virtual Join links a main data table to one or more auxiliary data tables. The feature enables the main data table to access data from the auxiliary data tables without physically joining the tables. See *Using JMP*.

The following example shows how to virtually join data tables and create distributions from the joined data:

```
dt1 = Open( "$SAMPLE_DATA/Pizza Profiles.jmp" );
dt1:ID << Set Property( "Link ID", 1 );// add Link ID and turn it on

dt2 = Open( "$SAMPLE_DATA/Pizza Responses.jmp" );

// add Link Reference to the Choice1, Choice2, and Choice columns
dt2:Choice1 << Set Property( "Link Reference", Reference Table(
    "$SAMPLE_DATA/Pizza Profiles.jmp" ) );
dt2:Choice2 << Set Property( "Link Reference", Reference Table(
    "$SAMPLE_DATA/Pizza Profiles.jmp" ) );
dt2:Choice << Set Property( "Link Reference", Reference Table(
    "$SAMPLE_DATA/Pizza Profiles.jmp" ) );

obj = dt2 << Distribution( // create a distribution of Crust[Choice1]
    Weight( :Subject ),
    Nominal Distribution(
```

```
        Column(
            Referenced Column(
                "Crust[Choice1]", // specify the joined column name
                Reference( Column( :Choice1 ), Reference( Column( :Crust ) ) )
            )
        )
    ),
    Nominal Distribution( Column( :Choice1 ) )
);
```

For another example of virtual join, search the Help > Scripting Index for "virtual join".

Specifying Linked Column Names

The Columns panel in a data table shows you the names of the linked columns. For example, open the Movie Inventory.jmp and Movie Rentals.jmp sample data tables to see the linked columns shown in Figure 9.12.

The column name of the source column is followed by the name of the referencing column in brackets. For example, Figure 9.12 shows the Name source column followed by the referencing column [Item Number] in brackets.

Figure 9.12 Linked Columns in Movie Rentals.jmp

You change the name of the virtually linked column name with `Set Title`. For example, the following script changes the name of the virtually linked column Rating[Item Number] to Movie Rating in the Distribution report.

```
dt1 = Open( "$SAMPLE_DATA/Movie Inventory.jmp" );
dt2 = Open( "$SAMPLE_DATA/Movie Rentals.jmp" );
dt2 << Distribution(
  Nominal Distribution(
    Column(
      Referenced Column( // specify the joined column name
        "Rating[Item Number]",
        Reference( Column( :Item Number ), Reference( Column( :Rating ) ) )
```

```
      )
     )
   ),
   SendToReport( Dispatch( {}, "Rating[Item Number]", OutlineBox,
      {Set Title( "Movie Rating" )} ) ) /* change the name of the
      virtually linked column */
);
```

Note that running the script automatically opens the referenced table (the one with the Link ID property), just like running any other platform script, before the referenced table is opened.

You can also specify that the referenced column name be used in linked columns. This option lets you define a shorter column name (for example, Rating instead of Rating[Item Number]). Include the Use Linked Column Name option:

```
dt1 = Open( "$SAMPLE_DATA/Movie Inventory.jmp" );
dt2 = Open( "$SAMPLE_DATA/Movie Rentals.jmp" );
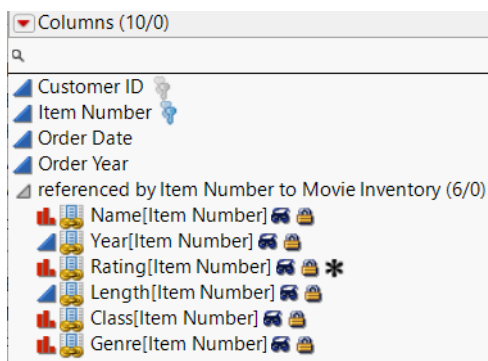
Column( dt2, "Item Number" ) << Set Property(
   "Link Reference",
   {Reference Table( "Movie Inventory.jmp" ), Options( "Use Linked Column
   Name"( 1 ) )}
);
dt2 << Distribution(
   Nominal Distribution(
      Column(
         Referenced Column( // specify the joined column name
            "Rating",
            Reference( Column( :Item Number ), Reference( Column( :Rating ) ) )
         )
      )
   )
);
```

Merging Referenced Data into One Data Table

To share the data from virtually joined data tables with another user, you might want to merge the data so that the data tables are permanently joined. This option is helpful because you provide one data table, not the main and auxiliary data tables.

When you merge referenced data, the data are replicated from the auxiliary data table to the main data table. When you save the main data table, the actual data are also saved.

The following example shows how to merge two referenced columns:

```
dt1 = Open( "$SAMPLE_DATA/Pizza Profiles.jmp" );
dt2 = Open( "$SAMPLE_DATA/Pizza Responses.jmp" );

dt1:ID << Set Property( "Link ID", 1 );
```

```
dt2:Choice << Set Property( "Link Reference", Reference Table( dt1 ) );
dt2:Choice1 << Set Property( "Link Reference", Reference Table( dt1 ) );
dt2:Choice2 << Set Property( "Link Reference", Reference Table( dt1 ) );

// select the columns to merge
dt2 << Select Columns( :Choice1, :Choice2 );

dt2 << Merge Referenced Data();
/* linked data from the referenced columns are now embedded in
Pizza Responses.jmp */
```

Note: If you don't select the columns to merge, all columns are merged.

Replace Data in Data Tables

Note: Merge Update() is an alias for Update().

Update() replaces data in one table with data from a second table.

```
dt << Update( // message to first table
  With( dataTable ), // the other data table
  By Row Number, /* default join type; alternative is
                  By Matching Columns( col1 == col2 ) */
  Ignore Missing, // optional, does not replace values with missing values
  );
```

To try this, make a subset of Big Class.jmp:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
NewHt = dt << Subset( Columns( :name, :height), Output Table Name( "hts" ) );
```

Next, add 0-6 inches to each student's height:

```
diff = Random Uniform( 0, 6 );
For Each Row( NewHt, :height = :height + diff );
```

Finally, update the heights of students in Big Class.jmp with the new heights from the subset table:

```
dt << Update(
  With( NewHt ),
  By Matching Columns( :name == :name),
  );
```

Controlling the Columns Added to an Updated Table

Your updated table might contain more columns than your original table. You can select
which columns are included in your updated table using the option Add Column from
Update Table().

To add no additional columns:

```
Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update Table( None )
);
```

To add some columns:

```
Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update table( :col1, :col2, :col3 )
);
```

Create a Table Using Tabulate

Tabulate constructs tables of descriptive statistics. The tables are built from grouping columns,
analysis columns, and statistics keywords. The following example creates a table containing
the standard deviation and mean for the height and weight of male and female students:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Tabulate( // message to data table
   Add Table( // start a new table
      Column Table( Grouping Columns( :sex ) ), // group using the column sex
      Row Table( // add rows to the table

         // use the height and weight columns for the analysis
         Analysis Columns( :height, :weight ),

         // show the standard deviation and mean
         Statistics( Std Dev, Mean )
      )
   )
);
```

You can apply a transformation to a column in the Tabulate table and set the format of the
column at the same time. Use the Transform Column() function inside Analysis Column().
For example, the following script applies the Log transformation to height and sets the column
format for the Mean and % of Total columns.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Tabulate(
  Set Format(
    Mean(
      :height( 10, 1 ),
      Analysis Column(
        Transform Column(
          "Log[height]",
          Formula( Log( :height ) )
        ),
        Format( 10, "Best" )
      )
    ),
    "% of Total"n(:height( 12, 2 ),
    Analysis Column(
      Transform Column(
        "Log[height]",
        Formula( Log( :height ) )
      ),
      Format( 12, 2 )
    ))
  ),
  Add Table(
    Column Table(
      Analysis Columns(
        :height,
        Transform Column(
          "Log[height]",
          Formula( Log( :height ) )
        )
      ),
      Statistics( Mean, "% of Total"n )
    ),
    Row Table( Grouping Columns( :sex ) )
  )
);
```

Note: You can also change the width of columns and perform additional operations within Tabulate. See the JMP Scripting Index in the Help menu.

Copy Data Tables

The most efficient way to copy a data table is with the Subset() function. The following example subsets all rows and columns into New Table.jmp.

```
sourceDT = Open( "$SAMPLE_DATA/Big Class.jmp" );
destDT = sourceDT << Subset( Columns(), Rows(), Output Table Name( "New
    Table.jmp" ) );
```

You can also use `Copy File()`.

```
Copy File( "$SAMPLE_DATA/Big Class.jmp", "c:/MyData/New Table.jmp" );
```

`Copy File()` is equivalent to copying the file on your computer. However, you have to consider whether the destination data table with the specified name already exists. And the source file has to be saved. If the source file is not saved, or the destination file already exists, the copy fails. It's safer to use the `Subset()` method shown above.

Find Missing Data Patterns

If your data table contains a lot of missing data in different columns, you might want to see whether there is a pattern in the distribution of the missing data relative to the different columns.

```
dt << Missing Data Pattern( // message to data table
    Columns( :miss ), // find missing data in this column
    Output Table( "Missing Data Pattern" ) // name the output table
);
```

Compare Data Tables

JMP can compare two open data tables and report the differences between data, scripts, table variables, column names, column properties, and column attributes. To compare data tables using JSL:

```
obj = dt << Compare Data Tables(
Compare With( Data Table ( "Data Table Name" ) );
```

For example, to compare the data tables Students1.jmp and Students2.jmp, proceed as follows:

```
dt = Open( "$SAMPLE_DATA/Students1.jmp" );
dt2 = Open( "$SAMPLE_DATA/Students2.jmp" );
obj = dt << Compare Data Tables( Compare With( Data Table( "Students2" ) ) );
```

To view the difference summary matrix for the comparison results, use this function:

```
mtx = ( obj << Get Difference Summary Matrix );
```

The resulting matrix appears in the log window:

```
[-1 1 2 2,
-1 2 4 3,
-1 1 7 4,
1 3 8 4,
1 1 10 9,
```

```
    0 1 11 11,
    -1 1 14 14,
    -1 1 16 15,
    1 1 18 16,
    -1 1 19 18,
    0 1 22 20,
    0 1 26 24,
    1 1 29 27,
    1 1 34 33]
```

In the comparison summary matrix in the example, the first column represents the action
performed on a column: -1 for a Delete, 0 for a Replace, and 1 for an Add. The second column
represents the number of rows affected by the action. The third and fourth columns represent
the row numbers affected in the two data tables in their respective order (Students1.jmp,
Students2.jmp).

Create a Summary Table

A Summary table includes summary statistics for a data table, such as the mean, median,
standard deviation, minimum value, and maximum value. By default, the summary table is
linked to its source table. When you select rows in the summary table, the corresponding rows
are highlighted in its source table.

```
dt << Summary( <private>, <invisible>, <Group( column )>, <Weight( column )>,
    <Freq( column )>, <N>, <Mean( column )>, <Std Dev( column )>, <Min( column
    )>, <Max( column )>...
```

Include columns for any statistic listed in the Summary launch window.

The following example creates a summary table from the Fitness.jmp sample data table. The N
Rows column, which specifies the number of rows for each level, is included in the Summary
table by default. This script creates a column of the mean for Runtime by Sex.

```
dt = Open( "$SAMPLE_DATA/Fitness.jmp" );
dt << Summary(
        Group( :Sex ),
        Mean( :Runtime ),
);
```

You can override the preselected Freq or Weight columns in the data table by specifying the
Freq or Weight columns in the script. To omit the preselected Freq or Weight, use the keyword
"none".

```
dt = Open( "$SAMPLE_DATA/Fitness.jmp" );
dt << Summary(
        Group( :Sex ),
        Mean( :Runtime ),
        Weight( "none" )
);
```

To unlink the Summary table, include `Link to Original Data Table( 0 )` in the script.

```
dt = Open( "$SAMPLE_DATA/Fitness.jmp" );
dt << Summary(
    Group( :Sex ),
    Mean( :Runtime ),
    Link to Original Data Table( 0 )
);
```

Subscribe to a Data Table

If you want to receive a message when a data table changes, use the Subscribe message. For example, you might want a message sent to the log when columns are added or deleted.

Here is the basic syntax:

```
dt << Subscribe( keyname(<client>), On Delete Columns|On Add Columns|On Add
    Rows|On Delete Rows|On Rename Column|On Close|On Save|On Rename(function)
    );
```

The first argument is a name for the subscription (or "client"), so that it can also be removed.

Note: The subscription is to the data table. The subscriber subscribes to the broadcast from the data table; it does not subscribe to a program or to script.

The application can also subscribe as a client to the data table (for example, most built-in platforms such as Distribution). If a data table has clients when the data table is closed, the user is warned that there are applications open that might need the data table.

Each subscription remains in effect until you unsubscribe:

```
dt << Unsubscribe("keyname", On Delete Columns | On Add Columns | On Add Rows
    | On Delete Rows | On Close | On Col Rename | All);
```

The following example sends messages to the log when a row is added or deleted:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
delRowsFn = Function( {a, b, rows},
    dtname = ( a << Get Name());
    Print( dtname );
    Print( b );
    Print Matrix( rows );
);
addRowsFn = Function( {a, b, insert},
    dtname = ( a << Get Name() );
    Print( dtname );
    Print( b );
    Print( insert );
);
```

```
dt << Subscribe( "Test Delete", onDeleteRows( delRowsFn, 3 ) );
dt << Subscribe( "Test Add", onAddRows( addRowsFn, 3 ) );
```

`Subscribe to Data Table List()` subscribes to a list of data tables to be notified when a new data table has been added, closed, or renamed. The following example shows how to subscribe and unsubscribe:

```
f2 = Function( {x}, Show( x ) );
Show( Subscribe to Data Table List( "My Data", OnClose( f2 ) ) );
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Wait( 2 );
Close( dt );
Show( Unsubscribe to Data Table List( "My Data" ) );
    Subscribe to Data Table List( "My Data", OnClose( f2 ) ) = "My Data";
    x = Data Table( "Big Class" );
    Unsubscribe to Data Table List( "My Data" ) = Empty();
```

Empty Application Names

If `Subscribe` is called with an empty application name, JMP generates a unique name that is returned to the caller. In the following example, appname2 is subscribed to the data table as a client.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
appname1 = dt << Subscribe( "", On Close( Print( "Closing Data Table" ) ) );
appname2 = dt << Subscribe(
  ""( "client" ),
  On Close(
    Function( {dtab},
        dtname = ( dtab << Get Name() );
        Print( dtname );
    )
  )
);
dt << Unsubscribe( appname1, On Close );
dt << Run Script( "Distribution" )
```

Move Data Between Matrices and Data Tables

For information about moving information between a matrix and a data table, see "Matrices and Data Tables" on page 214 in the "Data Structures" chapter.

Add Hyperlinks to Text Columns

You can define a text column as a column of hyperlinks using the Event Handler column property. For example, you can create a column of geographical data as hyperlinks to a Google map, or you might want to include links to a Wikipedia web pages.

Note: To learn more about the Event Handler column property, add the column property to a column that contains a hyperlink and study the property in the JMP interface. Tips about the property are provided. To see the script that creates the property, select the data table red triangle menu; select **Copy Script (No Data)**; and copy the script to an empty script.

Use the Event Handler column property to build hyperlinks based on one or more columns in your data table. This example shows how the hyperlinks in the SAS Offices.jmp sample data table were created.

1. Select **Help > Sample Data Library** and open SAS Offices.jmp.

2. Right-click the City column and select **Column Info**.

   Notice that the Event Handler column property has been added.

3. The script on the Click tab inserts the code for the URL.

   ```
   Function( {thisTable, thisColumn, iRow},
   Web(
       "https://www.google.com/maps/@"||Char(thisTable:latitude[irow])||","||Char
       (thisTable:longitude[irow])||",12z" ) );
   ```

   – `thisTable` is the data table object.

   – `thisColumn` is the column object.

   – `iRow` is the index of the row in `thisColumn`.

   – `Web` opens the URL in the default browser.

   – The @ sign in "https://www.google.com/maps/@" is specific to the Google API for maps. Google expects coordinates to follow it.

   – `||Char(thisTable:latitude[irow])` concatenates the latitude defined in the Latitude column.

   – `||","||Char(thisTable:longitude[irow])||` concatenates the longitude defined in the Longitude column.

   – The Google API uses `",12z"` to specify a zoom factor.

• The script on the Tip tab defines the tooltip that appears when you hover over the hyperlink. The tip accurately describes what happens after the link is clicked. Consider using this function to set the tooltip to the URL the Click script uses or a description of the platform.

   ```
   Function( {thisTable, thisColumn, iRow},
   ```

```
"Open " ||
   "https://www.google.com/maps/@"||char(thisTable:latitude[irow])||","||char
   (thisTable:longitude[irow])||",12z" || " in your browser.";);
```

– Returns the tool tip string.

• The script on the Color tab defines the color of the hyperlink (in this example, blue).

```
Function( {thisTable, thisColumn, iRow}, 5;);
```

See "JMP Colors" on page 454 for a diagram of JMP colors.

You might also want to embed links to other files in a data table column. The following example shows how to use Event Handler scripts and set the values to the audio and video locations on your computer.

```
New Table( "Wildlife",
   New Column( "Column 1", Character, "Nominal",
      Set Property( "Event Handler",
         Event Handler(
            Click( JSL Quote(Function( {thisTable, thisColumn, iRow}, Open(
   Char(
thisTable:thisColumn[ iRow ] ) ); );) ),
            Tip( JSL Quote(Function( {thisTable, thisColumn, iRow}, "Open " ||
   Char(
thisTable:thisColumn[ iRow ] ) || " in a media player."; );) ),
            Color( JSL Quote(Function( {thisTable, thisColumn, iRow}, 5; );) )
         )
      ),
      Set Selected,
      Set Values( {
"C:\Users\Public\Videos\Sample Videos\Wildlife.wmv",
"C:\Users\Public\Music\Sample Music\Kalimba.mp3"
      } )
   )
)
```

The SAS Country Office and Photo columns in SAS Offices.jmp also contain examples of Event Handler scripts.

Columns

This section covers actions that you can perform on columns in a data table, such as creating, grouping, setting and getting attributes and properties, and so on.

- "Send Messages to Data Column Objects"
- "Create Columns"
- "Create a New Column by Text Matching"
- "Add Several Columns at Once"
- "Group Columns"
- "Use Set Selected to Select Columns"
- "Rearrange and Move Columns"
- "Recode Columns"
- "Add a Column Switcher"
- "Compress Selected Columns"
- "Create Custom Functions, Transforms, and Formats"
- "Custom Transform Example"
- "Delete Columns"
- "Get Column Names"
- "Column Attributes"
- "Column Properties"

Note: JMP can display math symbols and Greek letters (controlled in Preferences in the Font category). This means that if you save a column (such as T square limits), the column name could either be "T Square Limits" (no special characters) or "T^2 Limits" (with special characters). Any reference using the column name must match the name exactly, or it fails.

Send Messages to Data Column Objects

Just as you send data table messages to a data table reference, you can send column messages to a reference to a data column object. The Column function returns a data column reference. Its argument is either a name in quotation marks, something that evaluates to a name in quotation marks, or a number.

```
Column( "age" );        // a reference to the age column
col = Column( 2 );      // assign a reference to the second column
```

Note: You can also use the syntax `"Profits ($M)"n` to evaluate to a name that contains special characters.

The *Scripting Guide* uses *col* to represent data column references. To see the messages that you can send to data column objects, see the Scripting Index (**Help > Scripting Index > Objects > Data Tables > Column Scripting**). Alternatively, you can use the `Show Properties()` command:

```
Show Properties( col );
```

Note: With column references, you must include a subscript to refer to the individual data values. Without a subscript, the reference is to the column object as a whole.

Once you have stored a data column reference in a global variable, to modify columns, you send messages to the data column reference.

Sending messages to columns is comparable to sending messages to data tables. Either state the object, a double-angle operator <<, and then the message with its arguments in parentheses, or use the `Send()` function with the object and then the message. In some cases the messages themselves need no arguments, so the trailing parentheses are optional.

```
col << message( arg, arg2, ... );
Send( col, message(arg, arg2, ... ) );
```

As with data tables and other types of objects, you can stack or list messages:

```
col << message << message2 << ...
col << {message, message2, ...};
```

Tip: To delete a column, you must send the message to a data *table* reference, because objects cannot delete themselves, only their containers can delete them.

Access Cell Values through Column References

Always use a subscript on a column reference to access the values in the cells of columns. Without a subscript, the reference is to the column object as a whole.

```
x = col[irow]     // specific row
x = col[]         // current row
col[irow] = 2;    // as an l-value for assignment
dt << Select Where( col[] < 14 );  // in a WHERE clause
```

Access Cell Values that have Value Labels

If the column whose values you want to access has a Value Label property, you might want to access the value label (formatted value) instead of the actual data value. To do this, use the formatted option. Consider the following example:

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Run Script( "Set Age Value Labels" );
x = Column( dt, "age", formatted )[1];
     Show( x );
y = Column( dt, "age" )[1];
     Show( y );
```

The log shows the formatted value for *x*, which is Twelve, and the actual data value for *y*, which is 12.

Create Columns

To add a new column to a data table, send a New Column message to a data table reference or use the New Column() function. The first argument, the column's name, is required. Either enclose the name in quotation marks or specify an expression evaluating to the name.

```
dt = New Table( "MyData1.jmp" );
dt << New Column( "wafer" );
```

or

```
dt = New Table( "MyData2.jmp" );
a = "wafer";
dt << New Column( a );
```

or

```
dt = New Table( "MyData3.jmp" );
col = New Column( "X", Formula( Random Uniform() ) );
```

If the table already includes a column by the same name, a sequential numeric value is appended to the new column name (wafer, wafer 2, wafer 3, and so on).

You can specify the following properties when creating a new column:

- Data type (numeric, character, row state, or expression)
- Modeling type (continuous, nominal, ordinal, multiple response, unstructured text, or vector)
- Column width (only for numeric columns)
- Numeric format

Unless otherwise specified, columns are numeric, continuous, and 12 characters wide.

The following example creates a new column whose data type is numeric, modeling type is continuous, width is set to 5, and numeric format is set to Best:

```
dt << New Column( "wafer", Numeric, "Continuous", Format( "Best", 5 ) );
```

The following example creates a character column and automatically assigns the nominal modeling type.

```
dt << New Column( "Last Name", Character );
```

You can also add formulas and other script messages appropriate to the column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Ratio", Numeric, "Continuous", Formula( :height/:weight )
   );
dt << New Column( "myMarkers",
   Row State,
   Set Formula( Marker State( age - 12 ) )
);
```

You can also create new columns that have the same properties as an existing column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Like Name", Like( :name ) );
```

If you plan to work with specific columns later (for example, grouping or changing data types), create a column reference:

```
myCol = dt << New Column( "Birth Date" );
```

Fill a Column with Data

To fill the column with data, use Values or its equivalent, Set Values. Include the value for each cell in a list.

The following example adds a new column called Last Name to a new data table. Three values are added: Smith, Jones, and Anderson.

```
dt = New Table( "My Data");
dt << New Column( "Last Name", Character, Values( {"Smith", "Jones",
   "Anderson"} ) );
```

The column can also be filled with numeric values. The following example adds a new column called Row Number to a new data table. The function N Row returns the number of rows in the table, and numeric values populate all of the rows in the column, beginning with 1.

```
dt = New Table( "My Data");
dt << New Column( "Last Name", Character, Values( {"Smith", "Jones",
   "Anderson"} ) );
dt << New Column( "Row Number",
   Numeric,
   Values( 1 :: N Row() ),
   Format( "Best", 5 )
);
```

To add a column of random numbers, insert a formula along with the random function.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Random", Numeric, Formula( Random Uniform() ) );
```

Another option is to add a constant numeric value to each cell. In the following example, the number 5 is added to each cell.

```
dt << New Column( "Number" );
:Number << Set Each Value( 5 );
```

New Column() can also be used as a built-in function, in other words, without the << (Send) command applied to the data table reference. When used in this way, the column is added to the current data table.

```
dt << New Column( "Address" ); // command is sent to the data table reference
New Column( "Address" ); // column is added to the current data table
```

Create a New Column by Text Matching

The New Column by Text Matching option enables you to use regular expressions (which identify patterns in text) to match text in a column and then create a new column from the matched text. This feature is an alternative to using the Text Explorer platform to write regular expressions and then save the results to a column.

The following example shows how to select *cat*, *cats*, *dog*, and *dogs* in a column and put the values in new columns.

```
dt = Open( "$SAMPLE_DATA/Pet Survey.jmp" );
dt << New Column by Text Matching(
  Column( :Survey Response ),
  Set Regex(
    Custom(
      Title( "Cat" ),
      /* finds "cat", optionally followed by "s",
         with either a space or period afterward. */
      Regex( "(cat(?:s?))([\s\.])" ),
      Result( "\[\1]\" ), // results in the value matched by (cat...)
    ),
  ),
  Output Column Name( "Instances of Cats" )
);
dt << New Column by Text Matching(
  Column( :Survey Response ),
  Set Regex(
    Custom(
      Title( "Dog" ),
      Regex( "(dog(?:s?))([\s\.])" ),
      Result( "\[\1]\" ),
    ),
```

```
    ),
    Output Column Name( "Instances of Dogs" )
);
```

For more information about using regular expressions in JSL, see "Regular Expressions" on page 172. To learn more about regular expressions, visit Regular-Expressions.info.

Add Several Columns at Once

The Add Multiple Columns message creates several columns at once. Arguments include the column name prefix, the number of columns, where to insert them (Before First, After Last, or After(col)), and the data type (Numeric, Row State, or Character(width)). An additional argument, field width, is optional for a numeric column.

```
// add two row state columns named "beginning <n>" before the first column
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Multiple Columns( "beginning", 2, Before First, Row State );

// add three numeric columns named "middle <n>" after the height column
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Multiple Columns( "middle", 3, After( :height ), Numeric );

// add four character columns named "end <n>" after the last column
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Multiple Columns( "end", 4, After Last, Character( 4 ) );
```

The column name is a prefix. If multiple columns with the same name are added, a sequential number is appended to each column name (beginning 1, beginning 2, and so on).

Group Columns

You can group columns by sending the data table the Group Columns message, which takes a list of columns to group as an argument. For example, the following code opens the Big Class.jmp sample data table and groups the age and sex columns.

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Group Columns( {:age, :sex} );
    "age etc."
```

You can also send a column name followed by the number of columns to include in the group. The group includes the first column named and the n-1 columns that follow it. This line is equivalent to the line above, which groups age and sex:

```
dt << Group Columns( :age, 2 );
```

The group name is based on the first column specified in the argument. So in the preceding example, the group is automatically named age etc. To customize the name, include the group name as the first argument.

```
dt << Group Columns( "My group", :age, 2 );
```

To ungroup grouped columns, use the Ungroup Columns message, which takes a list of columns in a group as an argument. For example, the following line ungroups the two columns grouped in the previous example.

```
dt << Ungroup Columns( {:age, :sex} );
```

Note the following about grouping and ungrouping columns:

- Both messages take a single list as an argument. The list must be enclosed in braces.

- You cannot create more than one group in a single message (for example, by giving the Group Columns message two lists of columns). Instead, you must send the data table two separate Group Columns messages.

- The Ungroup Columns message takes a list of columns to ungroup, not the name of a group of columns. You can remove a partial list of columns from a group. For example, this line creates a group of four columns:

```
dt << Group Columns( {:age, :sex, :height, :weight} );
```

And this line removes two of the columns, while leaving the other two in the group:

```
dt << Ungroup Columns( {:age, :sex} );
```

Notice that the column group's name does not change automatically. You have to change the name.

Get Column Groups or Names

To get column groups or names, use Get Column Group or Get Column Groups Names. The first part of this example creates two sets of grouped columns as shown in the previous section. Get Column Group() returns the column names in the specific group. Get Column Groups Names returns the names of the column groups.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Group Columns( {:age, :sex} );
   "age etc."
dt << Group Columns( {:weight, :height} );
   "height etc."
dt << Get Column Group( "age etc." );
   {:age, :sex}
dt << Get Column Groups Names();
   {"age etc.", "height etc."}
```

Select a Column Group

To select or deselect a column group, use the following messages:

```
dt << Select Column Group( "name" ); // select the columns in the group
dt << Deselect Column Group( "name" ); // deselect the columns in the group
```

The following example groups columns X and Y and groups columns Ozone through Lead, and then selects those groups in the data table:

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Group Columns( "xy", {:X, :Y} );
dt << Group Columns(
    "pollutants",
    :Ozone :: :Lead
);
dt << Select Column Group( "xy", "pollutants" );
```

Rename a Column Group

To rename a column group, use the following message:

```
dt << Rename Column Group( "name", "toname" );
```

The following example renames the column group from xy to coordinates:

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Group Columns( "xy", {:X, :Y} );
dt << Group Columns(
    "pollutants",
    :Ozone :: :Lead
);
Wait( 3 );
dt << Rename Column Group( "xy", "coordinates" );
```

Move a Column Group

To move a column group, use the following message:

```
dt << Move Column Group (To First | To Last | After(column) | After(group));
```

The following example moves the columns in the pollutants group to the end:

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Group Columns( "xy", {:X, :Y} );
dt << Group Columns(
    "pollutants",
    :Ozone :: :Lead
);
dt << Move Column Group( To Last, "pollutants" );
```

Use Set Selected to Select Columns

Send the Set Selected message to select a column.

```
col << Set Selected( 1 );
```

To select all columns with a specific string in the name, run the following script:

```
dt = Open( "$SAMPLE_DATA/Semiconductor Capability.jmp" );

colList = dt << Get Column Names( String );

For( i = 1, i <= N Cols( dt ), i++,
   If( Contains( colList[i], "NPN" ), /* select only columns with NPN
   in the name */
      Column( colList[i] ) << Set Selected( 1 )
   )
);
```

If you want to select many columns, consider sending the Select Columns message to the data table. See the *Scripting Guide*.

Get Selected Columns

To get a list of currently selected columns, use the Get Selected Columns message.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
:age << Set Selected( 1 );
:sex << Set Selected( 1 );
dt << Get Selected Columns();
   {:age, :sex}
```

Return the list of selected columns as a string using the string argument:

```
dt << Get Selected Columns( string );
   {"age", "sex", "height"}
```

Once you know what columns are selected, you can then write a script that acts upon these columns. Or your script can iteratively select columns and act upon them one at a time.

To actually select the columns before getting the columns, send the Set Selected message to a column. See "Column Attributes" on page 414.

Go To Column

To select and move to a specific column, use the Go To message.

```
dt << Go To( column name | column number );
```

For data tables with many columns, you can use this message to scroll the data table all the way to the left, so that the first column comes into view and is selected:

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
dt << Go To( 1 );
```

Invert Column Selections

To select any column that is currently deselected and deselect any column that is currently selected, use the `Invert Column Selection` message.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:age << Set Selected( 1 );
dt:height << Set Selected( 1 );
Wait( 1 );
b = dt << Invert Column Selection;
```

If a list of columns is provided, the columns that are not in the list are selected.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
a = {:height, :weight};
b = dt << Invert Column Selection( a );
```

Rearrange and Move Columns

These messages enable you to rearrange columns in a data table:

```
dt << Reorder By Name;          // alphanumeric order
dt << Reverse Order;            // reverse current order
dt << Reorder By Data Type;     // row state, character, and then numeric
dt << Reorder By Modeling Type; // continuous, ordinal, nominal
dt << Original Order;           // saved order
```

These commands move the currently selected column or columns to the indicated destination point.

```
dt << Move Selected Column( to first );
dt << Move Selected Column( to last );
dt << Move Selected Column( after( "name" ) );
```

Note that `Move Selected Column` is an alias for `Move Selected Columns`.

You can also move columns without first selecting them in the data table by using the following syntax.

```
dt << Move Selected Columns( {"name"}, to first );
dt << Move Selected Columns( {"name"}, to last );
dt << Move Selected Columns( {"name"}, after( "name" ) );
```

You may use a list to specify the column names.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
list = {"name", "sex"};
```

```
dt << Move Selected Columns( list, To Last );
```

Recode Columns

Use `Recode Column` to change all of the values in a column at once. Specify the original
column, or reference column, a list of transformations, and a target column. The function
applies the transformations to the reference column, updating the target column. If the
`Update Properties` argument is turned on, the following column properties are updated
when a column is recoded: Value Labels, Value Scores, Value Order, Value Colors,
Supercategories, Coding, and Missing Value Codes.

```
obj << Recode Column(<column reference>, {<transform>, ...}, <Update
    Properties(0|1)>, Target Column(<column reference> | <column name>))
```

For example, in Consumer Preferences.jmp, change the coding of the Gender column to 0|1
instead of 1|2, but keep the value labels the same.

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
col = New Column( :Gender );
dt << Recode Column(
    :Gender,
    {if(
        _rcNow == 1, 0,
        1,
    )},
    Update Properties( 1 ),
    Target Column( col )
);
```

You can also make the transformations to the original column. Recode the age column to age
groups in Big Class.jmp.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Recode Column(
    :age,
    {if(
        _rcNow >= 17, "Older",
        _rcNow >= 15, "Middle",
        "Younger"
    )},
    Target Column( :age)
);
```

Add a Column Switcher

Use JSL to add a Column Switcher panel to a report window. The Column Switcher lets you quickly analyze different variables without having to re-create your analysis.

```
// add a Column Switcher to the report
obj << Column Switcher(<default_col>(<col1>, <col2>, ...);

// remove the Column Switcher from a report
obj << Remove Column Switcher();
```

For example, add the Column Switcher to a Contingency report for the Car Poll.jmp data:

```
dt = Open( "$SAMPLE_DATA/Car Poll.jmp" );
obj = dt << Contingency(
    Y( :size ),
    X( :marital status )
);
ColumnSwitcherObject = obj << Column Switcher(
    :marital status,
    {:sex, :country, :marital status}
);
// number of pixels for the switcher width
ColumnSwitcherObject << Set Size( 200 );

// number of columns to display in the switcher
ColumnSwitcherObject << Set NLines( 6 );
```

You can also share a Column Switcher between two platform reports:

```
dt = Open( "$SAMPLE_DATA/Nicardipine.jmp" );
New Window( "Column Switcher Example",
    H List Box(
        sexCS = dt << Column Switcher( :Sex, {:Sex, :Causality, :Race} ),
        nDist = Distribution( Nominal Distribution( Column( :Sex ) ) ),
        bar = Graph Builder(
            Show Control Panel( 0 ),
            Variables( X( :Sex ), Y( :Age ) ),
            Elements( Box Plot( X, Y, Legend( 5 ) ) )
        )
    )
);

sexCS << Link Platform( nDist );
sexCS << Link Platform( bar );
```

Note: If you are publishing reports that share a Column Switcher to JMP Public or JMP Live, add them to a dashboard instead of using JSL. See *Using JMP*.

For more information about the Column Switcher, see *Using JMP*.

Compress Selected Columns

To minimize the size of a large data table, use the Compress Selected Columns message.
Each column is compressed into the most compact form.

```
dt << Compress Selected Columns( {column name, column name} );
```

For example, compress the age, sex, height, and weight columns in Big Class.jmp:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Compress Selected Columns(
{:age, :sex, :height, :weight}
);
```

For more information about this feature, see *Using JMP*.

Create Custom Functions, Transforms, and Formats

You can create user-defined custom functions and then use them in the Formula Editor's
function list, in JSL scripting, as custom formats, and as custom transforms. Note that these
custom functions can only be used in the context of the current JMP session. Restarting JMP
removes previously defined custom functions. They must be defined and added for each
session. An easy way to do this is to define your functions in an add-in or a startup script that
runs each time you start JMP.

To define a custom function, use New Custom Function(). After a function is defined, you
can add or activate it in the current running instance of JMP using Add Custom Functions().
Similarly, you can remove or deactivate a function using Remove Custom Functions().

New Custom Function() requires three parameters: namespace, functionName,
functionDefinition. Additional messages may be passed to define the following:

- Formula Editor values: Formula category, Result type, Parameters
- Custom transform values: Transform category
- Custom format values: Custom format category
- Scripting Index values: Description, Prototype, Scripting Index category, Examples

Tip: If you receive a data table that uses a custom transform, function, or format, you'll see
"###" in the column. The data table author must give you a script to add that custom item to
your computer. Run the script and then reopen the data table.

Custom Function Example

Let's create a completely specified function (including Scripting Index information and hint text). We will define the function first, then activate it.

```
funcDef = Function( {x, y = 10}, x + y );
description = "This function adds 2 values. If only 1 argument is specified,
   the 2nd argument defaults to 10.";

// examples can be multiple statements inside expr()
ex2 = Expr(
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
   myNameSpace:Add Ten( :age[4], 20 );
);
parmHint1 = "x value";
parmHint2 = "<y=10>";

/* create function using variables
the commands can also be accepted as arguments in New Custom Function */
newAdd = New Custom Function( "myNamespace", "Add Ten", funcDef );
newAdd << Description( description );
newAdd << Prototype( "myNameSpace:Add Ten(x,<y=10>)" );
newAdd << Example( "myNameSpace:Add Ten(4)" );

// expr() Examples need to be passed inside NameExpr()
newAdd << Example( Name Expr( ex2 ) );
newAdd << Parameter( "Number", parmHint1 );
newAdd << Parameter( "Number", parmHint2 );
newAdd << Formula Category( "NumberStuff" );

// add myNamespace:Add Ten function to system
Add Custom Functions( newAdd );
```

Running the above example will write "Deploying function: myNamespace:Add Ten" in the JMP log. Search for this function in the Scripting Index to see the results of using the Description, Prototype, and Example messages.

Figure 9.13 Custom Function in the Scripting Index

The `Formula Category` message results in a new category in the Formula Editor's list of formula categories. Custom functions are shown with an underline in the Formula Editor. Also see the hint text from the `Parameter` messages.

Figure 9.14 Custom Function in the Formula Editor

Delete a Custom Function Example

To delete a custom function, use `Remove Custom Functions( {"name"} )`.

```
Remove Custom Functions( {"myNamespace:Add Ten"} );
```

Custom Transform Example

Let's create a simple Custom Transform to multiply a value by 1,000. We embed the function definition inside `Add Custom Functions()`.

```
Add Custom Functions(
  New Custom Function(
     "custom", // namespace in which new function resides
     "x1000", /* function name. The completely scoped name is
        "custom:x1000".*/
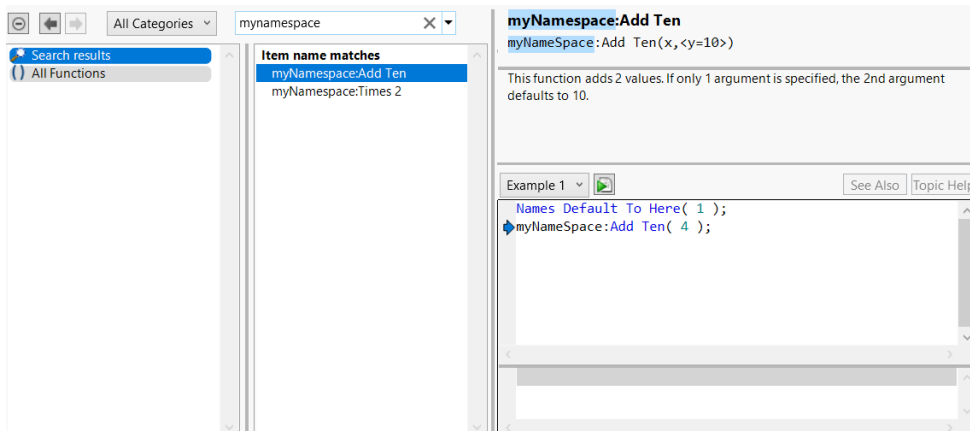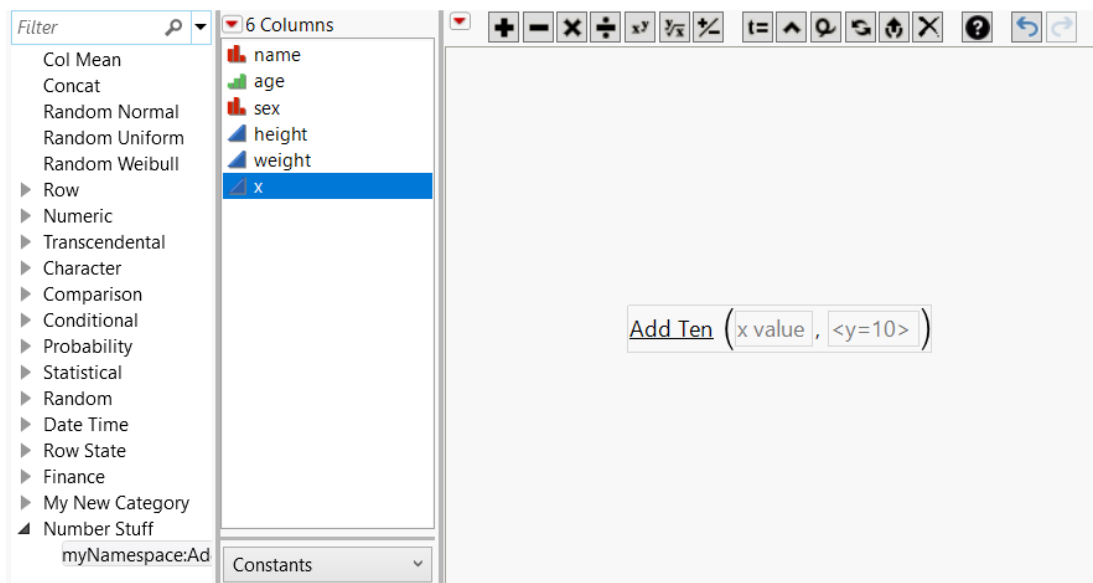
     Function( {x}, x * 1000 ),// function definition

     // optional message to enables this as a custom transform
     <<Transform Category( 1 )
     )
);
```

Running the above example writes "Deploying function: custom:x1000" in the JMP log. Search for this function in the Scripting Index to show that `custom:x1000` is an existing function.

When you create a new formula column in a data table, the custom transform appears as shown in Figure 9.15.

Figure 9.15 Custom Transform in a Data Table

Custom Format Example: Multiplication

The last type of custom function is a custom format. You create custom formats using `New Custom Function()` and by specifying the `Custom Format Category(1)` message. Here's an example that multiplies an input variable by 2 and also displays `(x2)` after the value. Note that, in this example, this function persists as a custom format and in the formula category "My New Category". This allows you to use the format in a column formula or as a custom format.

```
/* add a New Custom Function, specifying it as a Custom Format
show "(x2)" after the value in the data table */
Add Custom Functions(
  New Custom Function(
    "myNamespace",
    "Times 2",
    Function( {inputVar},
      {Default Local},
      Char( inputVar * 2 ) || " (x2)"
    ),
    <<Description( "Multiply input by 2. Display (x2) after the new value."
  ),
    <<Formula Category( "My New Category" ),
    <<Custom Format Category( 1 ) // enable this as a custom format
  )
);
```

By specifying the `Custom Format Category(1)` message, this format now persists under the Custom Function category in the Format menu. Here's what this looks like when applying this new Times 2 custom format in the Column Info window:

Figure 9.16 Custom Format in the Col Info Window

Custom Format Example: Dates

Suppose that you want to display dates in a format that isn't provided in JMP. The following script formats a data to show the month, year, day, and day of week.

```
Add Custom Functions(
    New Custom Function(
        "myNamespace",
        "DateYMD_Day",
        Function( {inputVar},
            {Default Local},
            Char(
                Match( Month( inputVar ),
                    1, "January ",
                    2, "February ",
```

```
                3, "March ",
                4, "April ",
                5, "May ",
                6, "June ",
                7, "July ",
                8, "August ",
                9, "September ",
                10, "October ",
                11, "November ",
                12, "December "
            )
        ) || Char( Year( inputVar ) ) || ", Day#" || Char( Day( inputVar ) )
    || "-" ||
        Char(
            Match( Day Of Week( inputVar ),
                1, "Sunday",
                2, "Monday",
                3, "Tuesday",
                4, "Wednesday",
                5, "Thursday",
                6, "Friday",
                7, "Saturday"
            )
        )
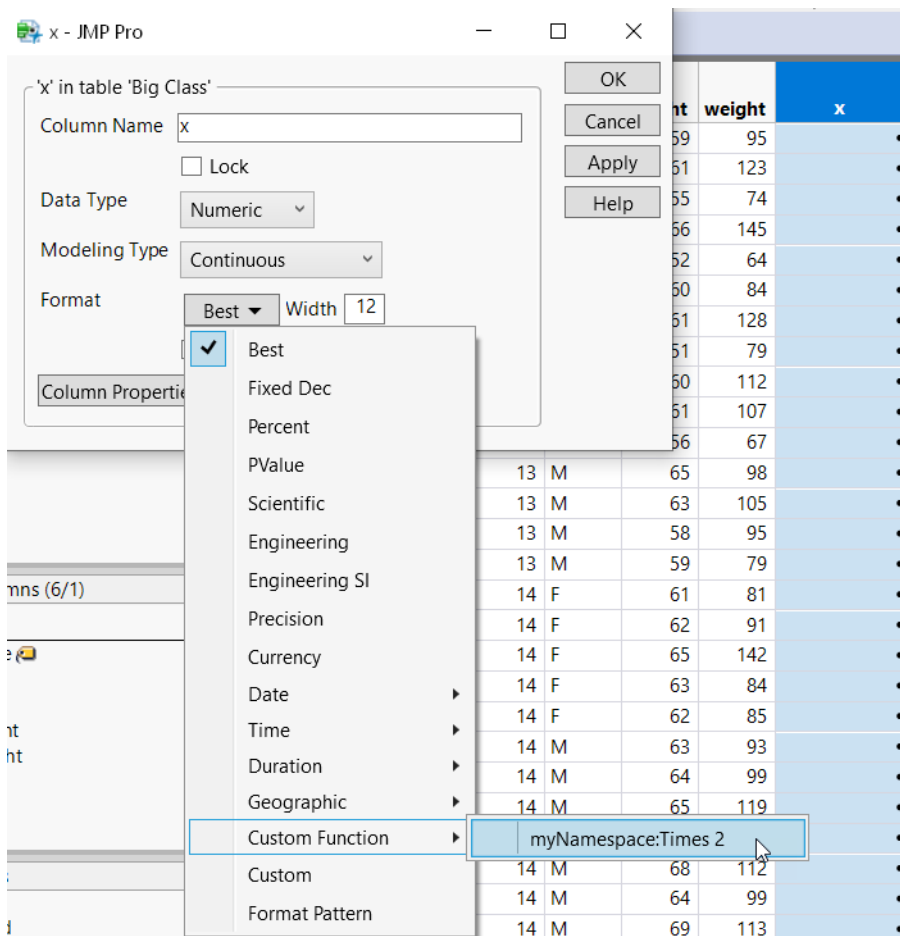
    ),
    <<Description(
        "Date format showing Month, Year, Day and Day of Week \!nfor example,
    April 1930, Day#29-Tuesday"
    ),
    <<Custom Format Category( 1 )
  )
);
```

After you run the script, you can assign the format to a column of date values. In the Cols >
Column Info window, the new date format is in the Format > Custom Function menu. You
need to widen the data table column to be wide enough to display all of the characters.

Tip: See "Display Numbers as Four Digits" on page 417 for another example of creating a
custom format.

Delete Columns

To delete columns, send a `Delete Columns` message and specify which column or columns to
delete. To delete more than one column, list the columns as multiple arguments or as a list.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Delete Columns( "weight" );
dt << Delete Columns( "weight", "age", "sex" );
dt << Delete Columns( {"weight", "age", "sex"} );
```

Without an argument, `Delete Columns` deletes columns that were previously selected. See "Column Attributes" on page 414.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column("age") << Set Selected;
dt << Delete Columns();
```

Get Column Names

`Column Name(n)` returns the name of the *n*th column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column Name( 2 );
    age
```

The returned value is a name value, not a quoted string. What this means is you can use it anywhere you would normally use the actual name in a script. For example, you could subscript it:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column Name( 2 )[1];
    12
```

If you want the name as a text string, quote it with `Char`:

```
Char( Column Name( 2 ) );
    "age"
```

To retrieve a list of the names of all columns in a data table, submit `Get Column Names`.

```
dt << Get Column Names( argument );
```

where the optional *argument* controls the output of the `Get Column Names` function, as follows:

- Specify `Numeric`, `Character`, `Row State`, or `Expression` to include only those column data types.
- Specify `Continuous`, `Ordinal`, `Nominal`, `Multiple Response`, `Unstructured Text`, `None`, and `Vector` to include only those modeling types.
- Specify `String` to return a list of strings rather than column names.

For example, if you want to get numeric and continuous columns in the Big Class.jmp sample data table, proceed as follows:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

```
names = dt << Get Column Names(Numeric, "Continuous")
  {height, weight}
```

Column Attributes

Use a collection of message pairs for data table columns to control all of the various attributes or characteristics of a column, including its name, data, states, and metadata. The messages come in pairs, one to "set" or assign each attribute and one to "get" or query the current setting of each attribute.

For example, you can hide, exclude, label, and turn on or off the scroll lock for a column through scripting. The value is Boolean; enter a one to turn the column attribute on, and a zero to turn it off.

In the following examples, the name column is unhidden, unexcluded, labeled, and locked from horizontal scrolling.

```
Column( "name" ) << Hide( 0 );
Column( "name" ) << Exclude( 0 );
Column( "name" ) << Label( 1 );
Column( "name" ) << Set Scroll Locked( 1 );
```

Note: All the messages to set various arguments (for example, Set Name, Set Values, Set Formula) start with Set. The word Set is optional for all messages except Set Name (recall that Name is already used for something else, the command that lets you use unusual characters in a name). Use whichever form you prefer or find easier to remember. The corresponding messages to retrieve the current value of an argument (for example, Get Formula) are the same, except that they start with Get instead of Set, and the word Get is *not* optional.

To deselect all selected columns, send a Clear Column Selection message to the data table object.

```
dt << Clear Column Selection;
```

Set or Get a Column Name

Set Name lets you name or rename a column, and Get Name returns the name for a column. The following example changes the name of the second column from age to ratio. It then returns the current column name to the log.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( 2 );
col << Set Name( "ratio" );
col << Get Name;
  "ratio"
```

Set or Get Column Values

Similarly, `Set Values` sets values for a column. If the variable is character, the argument should be a list. If the variable is numeric, the argument should be a matrix (vector). If the number of values is greater than the current number of rows, the necessary rows are added to the data table. `Get Values` returns the values in list or matrix form. `Get As Matrix` is similar to `Get Values` but returns values in the numeric columns.

```
col << Set Values( myMatrix ); // for a numeric variable
col << Set Values( myList ); // for a character variable
col << Get Values; // returns a matrix, or list if character
col << Get Matrix(<list of column names>|<list of column numbers>|<column
   range>; // returns the specified columns as a matrix
```

The following example returns a list and a matrix of values:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column( "name" ) << Values( {Fred, Wilma, Fred, Ethel, Fred, Lamont} );
myList = :name << Get Values;
   {"Fred", "Wilma", "Fred", "Ethel", "Fred", "Lamont", "JAMES", "ROBERT",
      "BARBARA", ...}

Column( "age" ) << Values( [28, 27, 51, 48, 60, 30] );
myVector = :age << Get Values;
   [28, 27, 51, ... ]
myMatrix = :weight << Get as Matrix;
   [95, 123, 74, ... ]
```

Set or Get Value Labels

Note: For more information about value labels, see *Using JMP*.

Value labels provide a method of displaying a descriptive label for abbreviated data. For example, you might have a column of 0 and 1 values, where 0 represents a male and 1 represents a female. The value label "male" for 0 and "female" for 1 are more readable.

You can specify value labels in any one of the following three ways. Using the Big Class.jmp sample data table, assume that M maps to Male, and F maps to Female.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
:sex << Value Labels( {"F", "M"}, {"Female", "Male"} ); // use two lists
:sex << Value Labels( {"F", "Female", "M", "Male"} ); // use a list of pairs
:sex << Value Labels( {"F" = "Female", "M" = "Male"} ); // use a list of
   assignments
```

You can activate value labels by sending `Use Value Labels` as a message to the column.

```
:sex << Use Value Labels( 1 );
```

To revert back to showing the column's actual values:

```
:sex << Use Value Labels( 0 );
```

The same message can be used for the data table to turn value labels on and off for all columns.

```
dt << Use Value Labels( 1 );
```

Set or Get Data and Modeling Types

You can set or get the data type of a column using JSL. The choices are character, numeric, and row state. The following example adds a new column that has a character data type to the Big Class.jmp sample data table.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "New" );
Column( "New" ) << Data Type( Character );
Column( "New" ) << Get Data Type;
    "Character"
```

To set or get the modeling type of a column:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "New" );
col << Modeling Type( "Continuous" );
col << Get Modeling Type;
    "Continuous"
```

You can specify the format of a column when changing its data type:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "Date" );
Column( "Date" ) << Data Type( Numeric, Format( "ddMonYYYY" ) );
```

Set or Get Formats

The Format message controls numeric and date/time formatting. The first argument is a quoted string from the list of format choices shown in the Column Info window. Subsequent arguments depend on the format choice. You can also set the field width by itself.

```
col << Format( "Best", 5 ); // width is 5
col << Format( "Fixed Dec", 9, 3 ); // width is 9, with 3 decimal places
col << Format( "PValue", 6 );
col << Format( "d/m/y", 10 );
col << Set Field Width( 30 );
```

For date formats, the Format message sets how dates appear in a data table column. To set the format that you use for entering data, or for displaying the current cell when you have it selected for entry or editing, use the Input Format message.

```
col << Format( "d/m/y", 10 ); // display the date in day-month-year order
col << Input Format( "m/d/y" ); // enter the date in month-day-year order
```

For more information about date/time formatting choices, see "Date-Time Functions and Formats" on page 150 in the "Types of Data" chapter.

Note: Do not confuse the Format *message* for columns with the Format *function* for converting numeric values to strings according to the format specified (typically used for date/time notation as described in "Date-Time Functions and Formats" on page 150 in the "Types of Data" chapter). Sending a message to an object has a very different effect from using a function that might happen to have the same name.

To get the current format of a column, submit a Get Format message:

```
col << Get Format;
```

Display Numbers as Four Digits

The Format() function and the Format message have a custom format capability that enables you to provide your own script to do the formatting. The name of the format is "Custom" (rather than "Best", "Fixed", and so on) and the argument is a script that works with the variable named "value" to return a string to represent that value. To make positive or negative integers always have at least four digits by adding zeros, you could do this:

```
New Table( "Untitled",
  New Column( "Column 1",
    Numeric,
    "Continuous",
    Format(
      "Custom", // make a custom format
      Formula( // using a JSL formula
        Local( {sign, length = 4, result}, // local variables
          sign = If( value < 0, -1, 1 ); // handle negative numbers
          value = Char( Abs( value ) );
          If( Length( value ) > length, // handle too big
            value = Repeat( "#", length )
          );
          result = Right( value, length, "0" ); // add zeros
          If( sign < 0,
            result = "-" || result // re-attach sign
          );
          result; // return the modified value
        )
      )
    ),
```

```
        // sample data to test positive, negative, zero
        Set Values( [0, 3, -5, 33, -55, 333, -555, 3333, -5555, 33333, -55555] )
    )
);
```

See "Create Custom Functions, Transforms, and Formats" on page 406 for more information about creating custom formats.

Specify the Significant Digits in a Numeric Column

Significant digits are the first nonzero digit and all succeeding digits. For example, 3.14159 has six significant digits, and 0.00314 has three significant digits.

Set the number of significant digits with the Precision column format:

```
Format( "Precision", 12, 3 ); // 12 wide with 3 significant digits
```

Set, Get, or Evaluate a Formula

The following examples show how to set, get, and evaluate a formula:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "Ratio" ); // create column and stores its reference
col << Set Formula( :height/:weight ); // set the formula
col << Eval Formula; // evaluate the formula
col << Get Formula; // return the expression :height:weight
```

To use the values from columns in scripts, be sure to add commands to evaluate the formula. Formula evaluation timing can differ between different versions of JMP.

Notes:

- When formulas are added, they are scheduled to be evaluated in a background task. This can be a problem for scripts if they depend on the column having the values while the script is running.

- To force a single column to evaluate, you can send an Eval Formula command to the column. You can do this inside the command to create the column, right after the Formula clause:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Ratio",
    Numeric,
    Formula( :height / :weight ),
    Eval Formula
);
```

where Formula() is an alias for Set Formula().

However, it is best to wait until you are finished adding a set of formulas, and then use the command Run Formulas to evaluate all of the formulas in their proper order, as follows:

```
dt << Run Formulas;
```

- The `Run Formulas` command is preferable to the `Eval Formula` command, because while it is evaluating the formulas, `Eval Formula` does not suppress the background task from evaluating them again. The formula dependency system background task takes great care to evaluate the formulas in the right order, and `RunFormulas` simply calls this task until all the formulas are finished evaluating.

- If you use random numbers and use the `Random Reset(`*seed*`)` feature to make a reproducible sequence, then you have another reason to use `Run Formulas`, in order to avoid a second evaluation in the background.

- If formula suppression is enabled with `Supress Eval`, turn it off to enable `Eval Formula` to run.

- The `Add Custom Function()` function enables you to create your own functions and display them in the Formula Editor's Functions list. See "Create Custom Functions, Transforms, and Formats" on page 406.

Set and Get Range and List Checks

You can manipulate list and range check properties using JSL. The following examples use the Big Class.jmp sample data table.

Set and clear the list check property in the sex column:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column( "sex" ) << List Check( {"M", "F"} ); // set the property
Column( "sex" ) << List Check(); // clear the property
```

Range checks require the specification of a range using the syntax in Table 9.3.

Table 9.3 Range Check Syntax

To specify this range	Use this function
$a \leq x \leq b$	LELE(a, b)
$a \leq x < b$	LELT(a,b)
$a < x \leq b$	LTLE(a,b)
$a < x < b$	LTLT(a,b)

The following example specifies that the values in the age column must be greater than zero and less than 120:

```
Column( "age" ) << Range Check( LTLT( 0, 120 ) );
```

All of the functions can be preceded by Not and one of them can be missing. The following
example specifies that the values in the age column should be greater than or equal to 12:

```
Column( "age" ) << Range Check( not( LT( 12 ) ) );
```

To clear a range check state, submit an empty Range Check():

```
Column( "age" ) << Range Check();
```

To retrieve the list or range check assigned to a column, send a Get List Check or Get Range
Check message to the column:

```
Column( "sex" ) << Get List Check;
Column( "age" ) << Get Range Check;
```

Here is an example of Get Range Check for the age column in Big Class.jmp:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column( "age" ) << Range Check( LTLT( 0, 120 ) );
Column( "age" ) << Get Range Check;
    Range Check( LTLT( 0, 120 ) )
```

Note that you can also use Set Property, Get Property, and Delete Property to set,
retrieve, and remove list checks and range checks. See "Column Properties" on page 421.

Note: Operations sent through JSL that involve range check columns show any warnings in
the log rather than in interactive windows.

Get a Column Script

Get Script returns a script to create the column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Column( "Ratio", Set Formula( :height/ :weight) );
Column( "Ratio" ) << Get Script;
    New Column( "Ratio",
        Numeric,
        "Continuous",
        Format( "Best", 10 ),
        Formula( :height / :weight )
    );
```

Preselect Roles

To preselect a role on a column, use the Preselect Role message. Choices include No Role,
X, Y, Weight, Freq, and Validation. The Get Role message returns the current setting.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "New" );
col << Preselect Role( X );
col << Get Role;
```

Lock a Column or Cells

To lock or unlock a column, use `Lock` or `Set Lock` with a Boolean argument. `Get Lock` returns the current setting.

```
col << Lock( 1 ); // lock
col << Set Lock( 0 ); // unlock
col << Get Lock; // show current state
```

Column Properties

Tip: JSL choices for properties are the same as those in the **Column Properties** menu in the Column Info window. The arguments for each property correspond to the settings in the Column Info window. An easy way to learn the syntax is to establish the property that you want in the Column Info window first, and then use `Get Property` to view the JSL. See *Using JMP* for more information about each property.

Data columns have numerous optional metadata attributes that can be set, queried, or cleared using the messages `Get Property`, `Set Property`, and `Delete Property`.

```
col << Set Property( "propertyName", {argument list} );
col << Get Property( "propertyName" );
col << Delete Property( "propertyName" );
```

The name of the property in question is always the first argument for `Set Property`, and what is expected for subsequent arguments depends on which property you set:

- `Get Property` and `Delete Property` always take a single argument, which is the name of the property.

- `Get Property` returns the property's settings. `Delete Property` completely removes the property from the column.

- To get a list of all column property names for a column, specify the column and then use `Get Properties List`.

If you want to set several properties, you need to send several separate `Set Property` messages. You can stack several messages in a single JSL statement if you want.

```
col << Set Property( "Axis",{Min(50), Max(180)} )<< Set Property( "Notes", "to
    get proportions" );
```

To get a property's value, send a `Get Property` message whose argument is the name of the property that you want:

```
Column("ratio") << Get Property( "axis" ); // return axis settings
```

To set columns as label columns:

```
dt << Set Label Columns( col1, col2, col3 );
```

To clear all label columns:

```
dt << Set Label Columns();
```

The same syntax works for Set Scroll Lock Columns, and Scroll Lock.

Set Column Properties with Variables

When writing JSL to script a column property, it is important to understand that the column is a repository for all column properties. The column stores the column property with the values supplied without evaluation or validation of the supplied arguments. In order to add column properties that contains variables using JSL code, the script must convert any variables to their actual values.

One way to do this is to use expressions. In the following example, the Eval Expr() function evaluates and replaces any variable wrapped in the Expr() function with its value. The outer Eval() function evaluates the entire statement after the replacement has occurred.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
// assign desired spec limits
lLimit = 55;
uLimit = 70;
tLimit = 62.5;

Eval(
    Eval Expr(

        // store the limits in the Spec Limits column property
        :height << Set Property("Spec Limits",
            {LSL( Expr( lLimit ) ), USL( Expr( uLimit ) ), Target( Expr( tLimit )
    ),
            Show Limits( 0 )}
        )
    )
);
```

Set and Get Column Properties

Table 9.4 shows examples for setting and getting popular column properties. See *Using JMP* for more information about each property.

Table 9.4 Common Properties for Data Table Columns

Property	Description	Example Using Arguments

Notes	Stores notes about a column. Is a quoted text string.	`col<<Set Property( "Notes", "Extracted from Fisher iris data" );`
		
		`col<<Get Property( "Notes" );`

List Check
Range Check
	Prescribes the possible values that can be entered in a column.	`col<<Set Property( List Check,{"F","M"} );`
		
		`col<<Set Property( "Range Check",LTLT(0, 120) );`
		
		`col<<Get Property( "List Check" );`
		
		`col<<Delete Property( "Range Check" );`

Missing Value Codes	Specifies column values that should be treated as missing.	`col<<Set Property( "Missing Value Codes", {0, 1} );`

Value Labels	Specifies labels to be displayed in place of the values.	`col<<Value Labels( {0 = "Male", 1 = "Female"} );`
		Use Boolean values to turn value labels on or off:
		`col<<Use Value Labels( 1 );.`

Value Order	Specifies the order in which you want the data to appear in reports.	`col<<Set Property( "Value Order", {"Spring", "Summer", "Fall", "Winter"} );`

Table 9.4 Common Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments

Value Colors and Color Gradient	Specifies colors for either categorical or continuous data, respectively.	Specify the color by assigning a JMP number to each value:
		
		`col<<Set Property( "Value Colors", {"Female" = 3, "Male" = 5} );`
		
		Specify a color theme:
		
		`col<<Set Property( "Value Colors", Color Theme( "White to Blue" ) );`
		
		If you omit the color theme, JMP uses the color theme preferences.
		
		Specify the gradient and the range and midpoint for Value Gradient:
		
		`col<<Set Property( "Color Gradient", {"White to Blue", Range( {18, 60, 25} )} );`

Axis	Most platforms use this (if it exists) when constructing axes. Mostly Boolean.	`col<<Set Property( "Axis",{Min(50), Max(180), Inc(0), Minor Ticks(10), Show Major Ticks(1), Show Minor Ticks(1), Show Major Grid(0), Show Labels(1), Scale(Linear)} );`

Units	Provided for custom uses. Specify the units of measure.	`col<<Set Property( "units", "grams" );`
		
		`col<<Get Property( "units" );`

Coding	Used for DOE and fitting. List with low and high values.	`col<<Set Property( "Coding", {59,172} );`
		
		`col<<Get Property( "Coding" );`

Mixture	Used for DOE, fitting, and profiling. Specifies the Mixture column properties in a list.	`col<<Set Property("Mixture", {0.2, 0.8, 1, L PseudoComponent Coding} );`
		
		`col<<Get Property( "Mixture" );`

Row Order Levels	Specify to sort levels by their occurrence in the data instead of by value.	`col<<Set Property( "Row Order Levels", 1 );`

Spec Limits	Used for capability analysis and variability charts.	`col<<Set Property( "Spec Limits", {LSL(-1), USL(1), Target(0)} );`
		
		`col<<Get Property( "Spec Limits" );`

Table 9.4 Common Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments

Control Limits	Used for control charts.	`col<<Set Property( "Control Limits", {XBar(Avg(44), LCL(29), UCL(69))} );`
		
		`col<<Get Property( "Control Limits" );`

Detection Limits	Set in DOE and used in generalized regression.	`col<<Set Property( "Detection Limits", {10, 50} );`

Response Limits	Set in DOE and used in desirability profiling.	`col<<Set Property( "Response Limits", {Goal("Match Target"), Lower(1,1), Middle(2,2), Upper(3,3)} );`
		
		`col<<Get Property("Response Limits");`
		
		Choices for `Goal` are `Maximize`, `Match Target`, `Minimize`, `None`. Other arguments take numeric value and desirability arguments.

Design Role	Used for DOE. Specify a single role.	`col<<Set Property( "Design Role", "Covariate" );`
		
		`col<<Get Property( "Design Role" );`
		
		Choices for role are `Continuous`, `Discrete Numeric`, `Categorical`, `Blocking`, `Covariate`, `Mixture`, `Constant`, `Uncontrolled`, `Random Block`, `Signal`, and `Noise`.

Factor Changes	Sets the difficulty of changing a factor (Easy, Hard, or Very Hard).	`col<<Set Property( "Factor Changes", Hard) ;`
		
		`col<<Get Property( "Factor Changes" );`

Sigma	Used for control charts. Specify known sigma value.	`col<<Set Property( "Sigma",1.332 );`
		
		`col<<Get Property( "Sigma" );`
		
		Each type of chart uses a different sigma calculation.

Distribution	Set the distribution type to fit to the column.	`col<<Set Property( "Distribution", Distribution( GLog ) );`
		
		`col<<Get Property( "Distribution" );`

Table 9.4 Common Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments

Time Frequency	Set the type of time frequency.	`col<<Set Property( "Time Frequency", Time Frequency( "Annual" ) );`
		
		`col<<Get Property( "Time Frequency" );`

Map Role	Set how the column is used to connect map shape data with name data. Specify the role and other information as necessary.	`col<<Set Property( "Map Role", Map Role( Shape Name Use( "filepath to data table", "column name" ) ) );`
		
		`col<<Get Property( "Map Role" );`

Supercategories	Groups specific categories into one category. Supported only in the Categorical platform.	`col<<Set Property( "Supercategories", {Group( "Genders", {"F", "M"} )} );`
		
		`col<<Get Property( "Supercategories" );`

Multiple Response	Specifies the character that separates the responses within a cell.	`col<<Set Property( "Multiple Response", Multiple Response( Separator( "." ) ) );`
		
		`col<<Get Property( "Multiple Response" ) );`
		
		Note: We recommend assigning the Multiple Response modeling type if the delimiter is a comma. Then you don't have to specify the delimiter in the column property. See *Using JMP*.

Table 9.4 Common Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments

Profit Matrix	Assigns weights that define a decision model.
	
	Specify a list that contains the weights in a matrix and categories in another list. In the matrix, each row contains weights that reflect the profit for predicting the category indicated by the column of the matrix.
	
	Caution: The matrix argument is the transpose of the profit matrix as displayed in the Column Properties window.
		`col<<Set Property( "Profit Matrix",`
		`{[5 -1, -1 4, -2 -2],`
		`{"F", "M", "Undecided"}} );`
		
		`col<<Get Property( "Profit Matrix" );`

Expression Role	Applies to columns that contain expressions. Specifies whether the expression should be interpreted as a Picture, a Matrix, or an Expression.	`col<<Set Property( "Expression Role",`
		`Expression Role( "Picture", MaxSize(`
		`640, 480 ), StretchToMaxSize( 1 ),`
		`PreserveAspectRatio( 1 ), Frame( 0 ) )`

[Custom property]	Provided for custom uses. Corresponds to **Column Properties > Other** in the Column Info window.
	
	The first argument is a name for the custom property, and the second argument is an expression.
		`col<<Set Property( "Date`
		`recorded",12Dec1999 );`
		
		`long date( col<<Get Property( "Date`
		`recorded") );`

Specify Engineering SI Units

The Engineering SI format displays the value with an appropriate SI suffix. For example, 3,100 is displayed as 3.1k. 0.0055 is displayed as 5.5m.

Table 9.5

Magnitude	Suffix	Magnitude	Suffix	Magnitude	Suffix	Magnitude	Suffix
10^3	k	10^{15}	P	10^{-3}	m	10^{-15}	f
10^6	M	10^{18}	E	10^{-6}	μ	10^{-18}	a
10^9	G	10^{21}	Z	10^{-9}	n	10^{-21}	z
10^{12}	T	10^{24}	Y	10^{-12}	p	10^{-24}	y

Rows

This section describes the messages for adding and manipulating the rows in a data table. Row messages are directed to a data table reference, and most act on the currently selected rows. A number of row messages might not be practical in scripting (for example, Move Rows).

- "Add Rows"
- "Delete Rows"
- "Select Rows"
- "Find Rows"
- "Move Rows"
- "Assign Colors and Markers to Rows"
- "Color Cells"
- "Hide, Exclude, and Label Rows"
- "Iterate on Rows in a Table"
- "Prevent Editing Rows"
- "Row States"

Add Rows

To add rows, send an Add Rows message and specify how many rows. You can also specify after which row to insert the new rows. The arguments can either be numbers or expressions that evaluate to numbers.

```
dt << Add Rows( 3 ); // add 3 rows to bottom of data table
```

```
dt << Add Rows( 3, 10 ); /* add 3 rows after the 10th row, moving the 11th and
    lower rows farther down */
```

A variation of Add Rows lets you specify an argument yielding a list of assignments.
Assignments can be separated with commas or semicolons.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Rows(
    {:name = "Peter", :age = 14, :sex = "M", :height = 61, :weight = 124}
);
```

or

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Rows(
    {:name = "Peter"; :age = 14; :sex = "M"; :height = 61; :weight = 124}
);
```

You can send several arguments yielding lists, or even a list of lists. The following script
creates a data table with Add Rows commands of each variety:

```
dt = New Table( "Cities" );
dt << New Column( "xx", Numeric );
dt << New Column( "cc", Character, width( 12 ) );

dt << Add Rows( {xx = 12, cc = "Chicago"} ); // single list
dt << Add Rows( // several lists
    {xx = 13, cc = "New York"}, {xx = 14, cc = "Newark"} );
dt << Add Rows( // list of lists
    {{xx = 15, cc = "San Francisco"}, {xx = Sqrt( 256 ), cc = "Oakland"}}
);
a = {xx = 20, cc = "Miami"};
dt << Add Rows( a ); // evaluate as single list

b={{xx = 17, cc = "San Antonio"},{xx = 18, cc = "Houston"}, {xx = 19, cc =
    "Dallas"}};
dt << Add Rows( b ); // evaluate as list of lists
```

Further information for rows can be specified with messages described under "Row States"
on page 441.

Delete Rows

To delete rows, send a `Delete Rows` message and specify which row or rows to delete. To delete more than one row, give a list or matrix as the *rownum* argument, or combine `Delete Rows` with other commands such as `For`. The *rownum* argument can be a number, list of numbers, range of numbers, matrix, or an expression that yields one of these. Without an argument, `Delete Rows` returns the number of selected rows and deletes those rows. When an argument is not specified or rows are not selected, `Delete Rows` only returns the number of selected rows: 0.

```
dt << Delete Rows( 10 ); // delete row 10
dt << Delete Rows( {11, 12, 13} ); // delete rows 11-13
myList = {11, 12, 13};
dt << Delete Rows( mylist ); // delete rows 11-13
dt << Delete Rows( 1 :: 20 ); // delete first 20 rows
dt << Delete Rows( [1 2 3] );  // delete first 3 rows
```

For example, the following script opens Big Class.jmp and deletes row 10:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Selected( Row State( 10 ) ) = 1; // select row 10
dt << Delete Rows;  // delete row 10
```

You can list duplicate rows, and you can list rows in any order with no consequence.

Here is a general way to remove the bottom *x* rows of a data table of any size. The following example removes five rows from the bottom of the data table:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
x = 5;
n = N Row( dt );
For( i = n, i > n - x, i--,
   dt << Delete Rows( i )
);
```

NRow counts the rows in the table. See

Select Rows

`Select All Rows` selects (or highlights) all of the rows in a data table.

```
dt << Select All Rows;
```

If all rows are selected, you can deselect them all by using `Invert Row Selection`. This command reverses the selection state for each row, so that any selected rows are deselected, and any deselected rows are selected.

```
dt << Invert Row Selection;
```

Note: With the exception of `Invert Row Selection`, whose result depends on the current selection, any new selection message starts over with a new selection. If you already have certain rows selected and you then send a new message to select rows, all rows are first deselected.

To select a specific row, use `Go To Row`:

```
dt << Go To Row( 9 );
```

To select specific rows in a data table based on their row number, use the `Select Rows` command. The argument to the command is a list of row numbers. For example, to select rows 1, 3, 5, and 7 of a data table:

```
dt << Select Rows( {1, 3, 5, 7} );
```

To select a range of rows, specify one of the following messages:

```
dt << Select Rows( Index( 7, 10 ) );
dt << Select Where( Any( Row() == Index( 7, 10 ) ) );
```

Both of these examples select rows seven through 10 in the current data table.

Select Where

To select rows according to data values, use `Select Where`, specifying a logical test inside the parentheses.

Tip: For a description of the functions and operators that you can use within a `Select Where` message, see "Operators" on page 103 in the "JSL Building Blocks" chapter.

For example, using the Big Class.jmp sample data table, select the rows where the students' age is greater than 13:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age > 13 );
```

Or, select the rows where the students' ages are less than 14:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( dt, 2 );
dt << Select Where( col[] < 14 );
```

The following example selects the rows where the student's ages are less than 15 and the sex is "F":

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age < 15 & :sex == "F" );
```

To select a row without deselecting a previously selected row, combine << `Select Where`
with << `Select Where` and the `Current Selection("extend")` argument. This is an
alternative to using an `OR` statement.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age == 14 );
dt << Select Where( :sex == "F", Current Selection( "extend" ) );
```

To select rows that contain specific strings, follow this example:

```
dt = Open( "$SAMPLE_DATA/Hollywood Movies.jmp" );
my_genres = {"Romance", "Comedy"};
dt << Select Where( Contains( my_genres, :Genre ) );
```

You can also create more complex conditions by using regular expressions.

```
// select rows where :name begins with "j" or "J".
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
/* \0 is a group reference that causes the expression to return the entire
   matched text. !Is Missing is needed because the regex returns the matched
   text if it successfully matched or missing if it failed to match. So !Is
   Missing means the string successfully matched */
dt << Select Where( !Is Missing( Regex( :name, "^j", "\0", IGNORECASE ) ) );

// select rows where :name ends with "N" or "E".
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( !Is Missing( Regex( :name, "(N|E)$", "\0" ) ) );
```

Select Excluded, Hidden, or Labeled Rows

To select rows that are currently excluded, hidden, or labeled:

```
dt << Select Excluded;
dt << Select Hidden;
dt << Select Labeled;
```

To select rows that are *not* excluded, hidden, or labeled, stack a select message and an invert
selection message together in the same statement, or send the two messages sequentially:

```
dt << Select Hidden << Invert Row Selection;
dt << Select Hidden;
dt << Invert Row Selection;
```

Select Duplicate Rows

To select duplicate rows in a data table, use the `Select Duplicate Rows` message:

```
dt = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
dt << Select Duplicate Rows(); // selects rows 301 and 8864
```

The second and subsequent duplicate rows are selected. Note that the duplicate values are case sensitive.

Note: By default, if no matching column is specified, all of the columns are matched.

In the preceding example, the duplicate rows might have been data input errors. Exclude the duplicate rows to remove them from the analysis:

```
dt = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
dt << Select Duplicate Rows();
dt << Exclude();
```

To select duplicate values in the rows of selected columns, specify the column names. The following example finds duplicate values in the rows of the Incident Number and Time columns.

```
dt = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
dt << Select Duplicate Rows( Match( Column( "Incident Number" ), :Time ) );
```

To select duplicate rows according to data values, select the duplicate rows, specify the data values, and then restrict the duplicate rows to the current selection.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Duplicate Rows( Match( :age ) );
dt << Select Where( :age > 15, Current Selection( "restrict" ) );
```

Refer to a Specific Cell in the Row

To refer to a specific cell, assign a subscript to the cell's row number. In the following example, the subscript [1] is used with the weight column. The formula then calculates the ratio between each height and the first value in the weight column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Column( "ratio", Formula( height / weight[1] ) );
```

Select Random Rows

To obtain a random selection, use this syntax:

```
dt << Select Randomly( number );
dt << Select Randomly( Sample Size( number ) );
dt << Select Randomly( probability );
dt << Select Randomly( Sampling Rate( probability ) );
```

These commands use a conditional probability to obtain the exact count requested.

Select Matching Cells

The row menu command Select Matching Cells is also implemented in JSL.

```
// select matching cells in the current data table
dt << Select Matching Cells;
// select matching cells in all open data tables
dt << Select All Matching Cells;
```

For more complicated selections, or to store selections permanently as row state data, see

Find Rows

`Get Rows` returns a matrix of rows that match the specified condition. The following example
select rows in which the age is greater than or equal to 16:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Get Rows Where( :age >= 16 );
   [35, 36, 37, 38, 39, 40]
```

`Get Selected Rows` returns a matrix of the currently selected rows. The following example
selects rows 1, 3, 5, and 7 and then returns the row numbers in a matrix:

```
dt << Select Rows( {1, 3, 5, 7} );
dt << Get Selected Rows;
   [1, 3, 5, 7]
```

Find Selected Rows

`Next Selected` and `Previous Selected` scroll the data table window up or down so that the
next selected row that is not already in view moves into view. The table wraps, so `Next
Selected` jumps from the bottom-most selected row to the top-most, and *vice versa* for
`Previous Selected`.

```
dt << Next Selected;
dt << Previous Selected;
```

Clear Selected Rows

To cancel a selection, leaving no rows selected, use `Clear Select`:

```
dt << Clear Select;
```

Move Rows

These commands move the currently selected rows to the indicated destination point.

```
dt << Move Rows( At Start );
dt << Move Rows( At End );
dt << Move Rows( After( rowNumber ) );
```

Assign Colors and Markers to Rows

You can use the `Colors` and `Markers` messages to assign (or change) colors and markers used for rows. These settings mostly affect graphs produced from the data table. Both messages expect numeric arguments to choose which color or marker to use. For more information about how numbers correspond to colors and markers, see "Colors and Markers" on page 451.

```
dt << Colors( 3 );  // set selected rows to red
dt << Markers( 2 ); // pick the X marker for selected rows
```

As with other row messages, you can stack selection and other messages together:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age == 12 ) // select the youngest subjects
   << Colors( 8 ) << Markers( 8 ); // and use purple open circles for them
```

`Color by Column` sets colors according to the values of a column that you specify, and `Marker by Column` works similarly:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Color by Column( :age );
dt << Marker by Column( :age );
```

Here are additional, named arguments:

- `Continuous Scale` (`Color by Column` only)   Assigns colors in a chromatic sequential fashion based on the values in the highlighted column.

- `Reverse Scale`   Reverses the color scheme in use.

- `Make Window with Legend`   Creates a separate window with a legend.

- `Excluded Rows`   Applies the row states to excluded rows.

- `Marker Theme`   Specifies the marker type.

- `Color Theme`   Specifies the color theme.

Color Cells

You can color individual cells in the data grid. For example, this line uses the row state color to color the cells:

```
dt << Color Rows by Row State;
```

You can also specify a color theme for either categorical or continuous columns:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
:height << Set Property(
  "Color Gradient",
  {"White to Blue", Range( 40, 80 )}
);
```

```
:height << Color Cell By Value( 1 ); // turn on the cell coloring

:age << Set Property(
  "Value Colors", // assign the value colors
  {12 = Red, 13 = Yellow, 14 = Green, 15 = Blue, 16 = Magenta, 17 =
  Gray}
);

:age << Color Cell By Value( 0 ); // turn off the cell coloring
```

You can also color specific cells. The following example sets rows 1, 5, 8 of the "name" column
to red:

```
:name << Color Cells( red, {1, 5, 8} );
```

To remove the color from specific cells, set the color to black. The following example removes
the color for row 1 of the "name" column.

```
:name << Color Cells( black, {1} );
```

You can color cells based on specific values. The following example colors the cells in the
height column. All cells containing a value greater than 60 are blue. All cells containing a value
equal to or less than 60 are purple.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For( i = 1, i <= N Row( dt ), i++,
  If( Column( dt, "height" )[i] > 60,
    Column( dt, "height" ) << Color Cells( 5, {i} ),
    Column( dt, "height" ) << Color Cells( 8, {i} )
  )
);
```

Note: The first argument for `Color Cells` represents the color value. The second argument
contains the row numbers.

Hide, Exclude, and Label Rows

Note: For more information about hiding, excluding, and labeling rows using row state
functions, see "Row States" on page 441.

Use the `Hide`, `Exclude`, and `Label` messages to hide, exclude, and label rows. These messages
are toggles, meaning that to turn the messages on you send them once, and to turn them off
you send the message a second time. They also accept Boolean arguments to explicitly turn
them off and on.

For example, to hide all rows in Big Class where age is greater than 13, you could do the following:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age > 13 );
dt << Hide( 1 );
```

Messages to the same object can be stacked together in a single statement, so you could simplify the preceding script.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age > 13 ) << Hide( 1 );
```

Iterate on Rows in a Table

In addition to built-in programming functions for iterating, JSL provides functions for iterating through data table rows, groups, or conditional selections of rows.

Generally, an expression is executed on the current row of the data table only. Some exceptions are the expressions inside formula columns, Summarize and the pre-evaluated statistics functions, and any use of data table columns by analysis platforms.

Set the Current Row

Note: The current row for scripting is *not* related to rows being selected (or highlighted) in the data table or to the current cursor position in the data table window. The current row for scripting is defined to be zero (no row) by default.

You can set the current row for a script using For Each Row or Row() = X.

```
Row() = 3; ...
For Each Row( ... );
```

For Each Row executes the script once for each row of the current data table. Note that Row()=1 only lasts for the duration of the script, then Row() reverts to its default value, which is zero. This means that submitting a script all at once can produce different results than submitting a script a few lines at a time.

Throughout this chapter, examples without an explicit current row should be assumed to take place within a context that establishes a current row. See

What is the Current Row?

By default, the current row number is 0. The first row in a table is row 1, so row 0 is essentially not a row. In other words, *by default, an operation is done on no rows*. Unless you take action to set a current row or to specify some set of rows, you get missing values due to the lack of data. For example, a column name returns the value of that column on the current row. Scope the column name with the prefix : operator to avoid ambiguity (to force the name to be interpreted as a column name).

```
:sex; // returns ""
:age; // returns .
```

Scoping names prevents you from getting a result that might look reasonable for the whole data table but is actually based on only one row. It also protects you from accidentally overwriting data values when making assignments to ambiguous names under most circumstances. You can have even more complete protection by using the prefix or infix : operator to refer specifically to a data column and the prefix :: operator to refer specifically to a global script variable. See "Advanced Scoping and Namespaces" on page 272 in the "Programming Methods" chapter.

You can use the Row() function to get or set the current row number. Row() is an example of an *L-value* expression in JSL: a function that returns its value unless you place it before an assignment operator (=, +=, and so on.) to set its value.

```
Row(); // returns the number of the current row (0 by default)
x = Row(); // store the current row number in x
Row() = 7; // make the 7th row current
Row() = 7; :age; // make the 7th row current and returns 12
```

Note that the current row setting only lasts for the portion of a script that you select and submit. After the script executes, the current row setting resets to the default (row 0, or no row). Therefore, a script submitted all at once can produce different results from the same script submitted a few lines at a time.

How Many Rows and Columns?

The N Rows() and N Cols() functions return the rows and columns in a data table.

```
N Rows( dt ); // number of rows
N Cols( dt ); // number of columns
```

N Rows and N Cols also count the number of rows in matrices. Note that NRow and NCol are synonyms. See the "Inquiry Functions" on page 208 in the "Data Structures" chapter.

Iterate a Script on Each Row

To iterate a script on each row of the current data table, put For Each Row around the script.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( If( :age > 15, Show( :age ) ) );
```

To specify the open data table, include a data table reference as the first argument.

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
For Each Row( dt1, If( :age > 15, Show( :age ) ) );
```

You can use For Each Row to set row states instead of creating a new formula column in the data table. The scripts below are similar, except that the first one creates a row state column, and the For Each Row script simply sets the row state without creating a column.

```
New Column( "My Row State", Row State, Formula( Color State( :age - 9 ) ) );
For Each Row( Color of( Row State() ) = :age - 9 );
```

To iterate a script on each row that meets a specified condition, combine For Each Row and If:

```
For Each Row( Marker of( Row State() ) = If( :sex == "F", 2, 6 ) );
```

You can use Break and Continue to control the execution of a For Each Row loop. See "Break and Continue" on page 124 in the "JSL Building Blocks" chapter.

Return Row Values

Dif() and Lag() are special functions that can be useful for statistical computations, particularly when working with time series or cumulative data.

- Lag() returns the value of a column *n* rows before the current row.
- Dif() returns the difference between the value in the current row and the value *n* rows previous.

The following lines are equivalent:

```
dt << New Column( "htDelta" );
For Each Row( :htDelta = :height - Lag( :height, 1 ) );
For Each Row( :htDelta = Dif( :height, 1 ) );
```

Add Sequence Data

Sequence() corresponds to the Sequence function in the Formula Editor and is used to fill the cells in a data table column. It takes four arguments and the last two are optional:

```
Sequence( from, to, stepsize, repeat );
```

From and to are not optional. They specify the range of values to place into the cells. If from = 4 and to = 8, the cells are filled with the values 4, 5, 6, 7, 8, 4, ...

Stepsize is optional. If you do not specify a stepsize, the default value is 1. Stepsize increments the values in the range. If stepsize = 2 with the above from and to values, the cells are filled with the values 4, 6, 8, 4, 6, ...

Repeat is optional. If you do not specify a Repeat, the default value is 1. Repeat specifies how many times each value is repeated before incrementing to the next value. If repeat = 3 with the above from, to, and stepsize values, the cells are filled with the values 4, 4, 4, 6, 6, 6, 8, 8, 8, 4, ... If you specify a Repeat value, you must also specify a Stepsize value.

The sequence is always repeated until each cell in the column is filled.

```
dt = New Table( "Sequence Example" ); // create a new data table
dt << New Column( "Count to Five" ); // add two columns
dt << New Column( "Count to Seventeen by Fours" );
dt << Add Rows( 50 ); // add fifty rows

/* fill the first column with the data sequence 1, 2, 3, 4, 5, ...
fill the second column with the data sequence 1, 1, 5, 5, 9, 9, 13, 13, 17,
   17, ... */
For Each Row (
   Column( 1 )[ ] = Sequence( 1, 5 );
   Column( 2 )[ ] = Sequence( 1, 17, 4, 2 );
);
```

Because Sequence() is a formula function, you can also set a column's formula to use Sequence() to fill the column. The following example creates a new column named Formula Sequence and adds a formula to it. The formula is a sequence that fills the column with values between 25 and 29, incremented by 1, and repeated twice (25, 25, 26, 26, 27, 27, 28, 28, 29, 29, 25, ...).

```
dt = New Table( "Formula Sequence Example" );
dt << Add Rows( 20 );
dt << New Column( "Formula Sequence", Formula( Sequence( 25, 29, 1, 2 ) ) );
```

The following are more examples of Sequence() results:

- Sequence(1,5) produces 1,2,3,4,5,1,2,3,4,5,1, ...
- Sequence(1,5,1,2) produces 1,1,2,2,3,3,4,4,5,5,1,1, ...
- Sequence(10,50,10) produces 10,20,30,40,50,10, ...
- 10*Sequence(1,5,1) also produces 10,20,30,40,50,10, ...
- Sequence(1,6,2) produces: 1,3,5,1,3,5, ... The limit is never reached exactly.

Note: If you want a matrix of values, then use the Index function, not Sequence.

Prevent Editing Rows

To prevent adding or deleting rows, use `Set Edit Lock`:

```
dt << Set Edit Lock( "add rows", "delete rows");
```

See "Prevent Changes to a Data Table" on page 362 for more information about preventing other data table operations.

Row States

There is a special data element type called a row state that stores various attributes in the data table. Row states can indicate the following:

• Whether a row is selected, excluded, hidden, or labeled

• Which marker type, color, shade, and hue to use for graphs

In JSL, you can use row state functions to manipulate row states.

About Row States

Row states change how JMP works with your data. Table 9.6 explains each row state. Remember that you can use several row states at once to get the combination of effects that you want.

Table 9.6 Row States

Row states	How they affect results
Excluded	If rows are excluded, JMP omits them from calculations for statistical analyses (text reports and charts). Results are the same as if the data was not entered. However, points are still included in *plots*. (To omit points from plots, use `Hide`. To omit points from all results, use both `Exclude` and `Hide`.)
Hidden	If rows are hidden, JMP does not show them in plots. However, the rows *are* still included in text reports and *charts*. (To omit points from reports and charts, use `Exclude`. To omit points from all results, use both `Exclude` and `Hide`.)
Labeled	If rows are labeled, JMP places row number labels, or the values from a designated Label column, on points in scatterplots.

Table 9.6 Row States *(Continued)*

Row states	How they affect results
Color	If rows have colors, JMP uses those colors to distinguish the points in scatterplots.
Marker	If rows have markers, JMP uses those markers to distinguish the points in scatterplots.
Selected	If rows are selected, JMP highlights the corresponding points and bars in plots and charts.

Note: For row state messages, omitting the argument *toggles* the option: if the option is off, the message turns it on, and if the option is on, the message turns it off. You can also use the `"toggle"`, `"switch"`, or `"flip"` argument, as in `r << Exclude( "toggle" )`.

About Row State Functions

When you give a row state function a number as its argument (or an expression that evaluates to a number), the function interprets that number as an index to its possible values.

Table 9.7 shows a comparison chart of the different row state functions, so that you can see which functions convert row states to numbers and numbers to row states. It also includes the numbers that you can use with each function.

Table 9.7 Row State Functions

Numbers	Convert from numbers to row states ⟶	Row states	Convert from row states to numbers ⟶	Numbers
1 or 0	`Excluded State(`*n*`)`	Excluded	`Excluded(`*rowstate*`)`	1 or 0
1 or 0	`Hidden State(`*n*`)`	Hidden	`Hidden(`*rowstate*`)`	1 or 0
1 or 0	`Labeled State(`*n*`)`	Labeled	`Labeled(`*rowstate*`)`	1 or 0
1 or 0	`Selected State(`*n*`)`	Selected	`Selected(`*rowstate*`)`	1 or 0
0 to 31	`Marker State(`*n*`)`	Marker	`Marker Of(`*rowstate*`)`	0 to 31

Table 9.7 Row State Functions *(Continued)*

Numbers	Convert from numbers to row states ➜	Row states	Convert from row states to numbers ➜	Numbers
0 to 84 (0–15 basics, 16–31 dark, 32–47 light, 48–63 very dark, 64–79 very light, 80–84 grays)	`Color State(n)`	Color	`Color Of(rowstate)`	0 to 84 (0–15 basics, 16–31 dark, 32–47 light, 48–63 very dark, 64–79 very light, 80–84 grays)
0–11 (rainbow order)	`Hue State(n)`	Hue		
–2 to 2 (dark to light)	`Shade State(n)`	Shade		

Assign Row States

To assign a row state using JSL, use `Select Where` to indicate which rows are affected, and then specify which row state to assign to the rows. In the following example, marker type 5 (a triangle) is assigned to rows in which sex is "F".

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :sex == "F" ) << Markers( 5 );
```

You can also use a formula to assign row states. Here is how you would do it in the data table:

1. Create a row state column.

2. Add a formula to the column that assigns a marker to each row in which sex is "F".

3. In the Column panel, right-click the star next to the row state column and select **Copy to Row States**.

The JSL equivalent would be something like this:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Row State",
    Row State,
    Set Formula( If( :sex == "F", Marker State( 4 ) ) )
);
Column( "Row State" ) << Copy To Row States();
```

Store Row State Information

A row state column is a dedicated column that *stores* row state information, but does not put the information into effect. You can then use For Each Row to put the row state column into effect. For more information about row state columns, see *Using JMP*.

The following example creates a row state column then puts the row states into effect:

1. Submit this line to start a new data table:

```
dt = New Table( "Row State Testing" );
```

2. Submit these lines to add row states to a column and add ten rows:

```
dt << New Column( "Row State Data", Row State, Set Formula( Color State( Row()
    ) ) );
dt << Add Rows( 10 );
```

3. Submit this line to put the row states into effect:

```
For Each Row( Row State() = :Row State Data );
```

Figure 9.17 Table with No Row States (left) and Table with Row States (right)

This action replaces any row states in effect with the row state combination from the row state column. For more information about how to change selected attributes of a row state without changing or canceling others, see "Set One Characteristic and Cancel Others" on page 449.

Set or Get Row States

From JSL, you can set or get row states directly using the Row State function. You can set or get the state for row *n* with Row State(n). If you do not supply an argument, you set or get the current row. To work with all rows, either use a For Each Row loop or work with formula columns.

To set a row state, place the row state expression on the left of an assignment (L-value):

```
Row State( 1 ) = Color State( 3 ); // make row 1 red
Row() = 8; Row State() = Color State( 3 ); // make the 8th row red
For Each Row( Row State() = Color State( 3 ) ); // make each row red
```

To specify the data table, include a data table reference as the first argument:

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
Row State( dt1, 1 ) = Color State( 3 ); // make row 1 red;
```

To get a row state, place the row state expression on the right side of an assignment:

```
x = Row State( 5 );  // put the row state of row 5 into x
x = Row State();   // row state of current row
```

Notes on Setting Row States

Be careful whether you set every aspect of Row State() or just one aspect of it, such as Color Of(Row State()). To see how this works, first color and mark all the rows:

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row(
   Row State() = Combine States( Color State( Row() ), Marker State( Row() ) )
   );
```

And now observe the difference between setting one attribute of a row state:

```
Color Of( Row State( 1 ) ) = 3; // make row 1 red without changing marker
```

And setting every aspect of a row state to a single state:

```
Row State( 1 ) = Color State( 5 ); // make row 1 blue and remove its marker
```

To copy all the current row states into a row state column:

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row(
      dt1,
      Row State() = Combine States(
            Color State( Row() ),
            Marker State( Row() )
         )
);
```

```
    dt1 << New Column( "rscol", "Row State", Set Formula( Row State() ) );
```

To copy several but not all of the current row states into a row state column, use a script like the following (commenting out or omitting any states that you do not want):

```
New Column( "rscol2",
  Set Formula(
    Combine States(
      Color State( Color Of() ),
      Excluded State( Excluded() ),
      Hidden State( Hidden() ),
      Labeled State( Labeled() ),
      Marker State( Marker Of() ),
      Selected State( Selected() )
    )
  )
);
```

To set a component of a row state:

```
Color Of( Row State( i ) ) = 3; // change color to red for row i
Selected( Row State( i ) ) = 1; // select the ith row state and set to 1
```

You can see from the example above that some of the functions convert numbers into states, and others convert states into numbers. Here are some helpful hints for remembering which are which:

Number-to-state functions have the word "State"   Functions that take number arguments and either return states or accept state assignments all have the word "State" in their names: `Row State`, `As Row State`, `Color State`, `Combine States`, `Excluded State`, `Hidden State`, `Hue State`, `Labeled State`, `Marker State`, `Selected State`, `Shade State`.

State-to-number functions are one word or have the word "Of"   Functions that take row state arguments (and assume that the argument `Row State()` if none is given) and functions that return or are set to numbers are either one word, or their second word is "Of": `Color Of`, `Excluded`, `Hidden`, `Labeled`, `Marker Of`, `Selected`.

Table 9.7 on page 443 is a helpful comparison chart for these functions.

The following lines are equivalent to their interactive commands:

```
Copy From Row States
Add From Row States
Copy To Row States
Add To Row States
```

Examples of Getting Row States

For example, create the following table containing row states:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Row State Data", Row State, Set Formula( Color State( Row()
   ) ) );
dt << Add Rows( 10 );
For Each Row( Row State() = :Row State Data );
```

To get row states, submit the following lines:

```
Row State( 1 ); // returns row state for row 1
Row() = 8; Row State();  // returns row state for current (8th) row
For Each Row( Print( Row State()) ); // returns the row states for each row
```

To get a row state and store it in a global, place Row State() on the right side of an assignment. Add the following lines to the preceding script:

```
::x = Row State( 1 ); // put the row state of row 1 in x, a global
Row() = 8; ::x = Row State();  // put the row state of the 8th row in x
Show( x ); // returns x = Color State( 8 )
dt << New Column( "rscol", Row State) ;
// put row states in rscol (a row state column)
For Each Row( :rscol = Row State() );
```

To get a component of a row state, place the row state expression on the right side of an assignment and also use one of the L-value operators:

```
x = Selected( Row State() ); // selection index of current row selected
```

Get and Set Multiple Characteristics at Once

You can get or set many characteristics at once by combining state settings inside Combine States(). You can also get or set each characteristic one at a time, the ultimate row state being the accumulation of characteristics. The following example sets green Y markers for males, but hides them in plots for now. It also sets red X markers for females but shows them in plots.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row(
  If( sex == "M",
     /* then */ Row State() = Combine States(
        Color State( 4 ), Marker State( 6 ), Hidden State( 1 ) ),
     /* else */ Row State() = Combine States(
        Color State( 3 ), Marker State( 2 ), Hidden State( 0 ) ) ) );
```

Get the row state for one row, such as the 6th:

```
Row State( 6 );
    Combine States(Hidden State(1), Color State(4), Marker State(6))
```

Notice that JMP returns a Combine State() combination. This is because a row state datum is not just the state of one characteristic, such as color, but the cumulative state of *all* the characteristics that have been set: exclusion, hiding, labeling, selection, markers, colors, hues, and shades. A list of such characteristics is called a *row state combination*.

Just as there can be many row state characteristics in effect, a row state *column* can have multiple characteristic row states as its values.

Set One Characteristic and Cancel Others

In addition to the overall `Row State()` function for getting or setting all the characteristics of a row state, there are separate functions to get or set one characteristic at a time preemptively. That is, to give a row one characteristic, canceling any other characteristics that might be in effect. The functions that set one characteristic and cancel others are `Color State`, `Combine States`, `Excluded State`, `Hidden State`, `Hue State`, `Labeled State`, `Marker State`, `Selected State`, `Shade State`.

For example, to make row 4 be hidden only:

```
Row State( 4 ) = Hidden State( 1 );
```

Set or Get One Characteristic at a Time

A row state is not just one characteristic, but many. To work with just one characteristic at a time, use one of the L-value operators with `Row State` on either side of the equal sign. The side of the equal sign depends on whether you want to get or set a characteristic. There is an L-value operator for each of the following characteristics: `Color Of`, `Excluded`, `Hidden`, `Labeled`, `Marker Of`, `Selected`.

This example hides row 4 without affecting any other characteristics:

```
Hidden( Row State( 4 ) ) = 1
```

This example stores the color of row 3 without getting any other characteristics:

```
::color = Color Of( Row State( 3 ) );
```

Identify Row State Changes

The `MakeRowStateHandler` message (sent to a data table object) obtains a callback when the row states change. For example,

```
f = Function( {X}, Show( x ) );
obj = dt << Make Row State Handler( f );
```

Then when you select a group of rows, the row numbers of any row whose row state changed are sent to the log. For example:

x:[3, 4, 28, 40, 41]

When a group is highlighted, it might call the handler twice, once for rows whose selection is cleared, then again for the new selection.

Exclude, Hide, Label, and Select

This section discusses conditions that have Boolean states (either on or off). These conditions include: excluding, hiding, labeling, and selecting rows.

- `Excluded` gets or sets an excluded index. The index is 1 for true or 0 for false, indicating whether each row is excluded.

- `Hidden` gets or sets a hidden index, which is 1 for hidden or 0 for not hidden.

- `Labeled` gets or sets a labeled index, which is 1 for labeled or 0 for not labeled.

- `Selected` gets or sets a selected index, which is 1 for selected or 0 for not selected.

The following examples illustrate these conditions:

```
Excluded( Row State() ); // returns 1 if current row is excluded, 0 if not
Hidden();                // returns 1 if current row is hidden, 0 if not
Labeled( Row State() );  // returns 1 if current row is labeled, 0 if not
Selected();              // returns 1 if current row is selected, 0 if not

Excluded( Row State() ) = 1; // exclude current row
Hidden() = 0;                // unhide current row
Labeled( Row State() ) = 1;  // label current row
Selected() = 0;              // deselect current row
```

Remember that these functions assume the argument `Row State()` if none is given.

`Excluded State`, `Hidden State`, `Labeled State`, and `Selected State` do the reverse; they get or set a row state condition as true or false according to the argument. Nonzero values set the row state to true, and zero values set it to false. Missing values result in no change of state.

```
Row State() = Excluded State( 1 ); // change current row state to excluded
Row State() = Hidden State( 0 );   // change current row state to not hidden
Row State() = Labeled State( 1 );  // change current row state to labeled
Row State() = Selected State( 0 );
// change current row state to not selected
```

Notice that the first two expressions above replace the row state with just the exclusion or just the unhiding, so that any preexisting row state characteristics are lost. More commonly, you would issue the `state` commands for all of the characteristics that you want inside a `Combine States`. The following example changes the third row to hidden and a green square marker:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Row State( 3 ) = Combine States(
  Color State( 4 ),
  Marker State( 3 ),
  Hidden State( 1 ) );
```

Another common way to use a `-State` command would be in a row state data column whose values could be added to the row state (for cumulative characteristics). The following example excludes each odd numbered row:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "myExcl",
    Row State,
    Set Formula( Excluded State( Modulo( Row(), 2 ) ) )
);
For Each Row( Row State() = Combine States( :myExcl, Row State() ) );
```

Clear Row States

`Clear Row States()` removes row states from the data table. The following script assigns a row state to each row and then removes the row state.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( Row State() = Color State( Row() ) );
Wait( 3 ); // pause for demonstration purposes
dt << Clear Row States();
```

To clear row states for selected rows, select the rows first and then clear the selected row states.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( Row State() = Color State( Row() ) );
dt << Select Rows( [ 5 ] );
Wait( 3 ); // pause for demonstration purposes
dt << Clear Selected Row States();
```

Colors and Markers

This section discusses conditions that have many choices. These conditions include: coloring, adding markers, different hues, or different shades to rows.

`Color Of()` returns or sets the color index. The color index is a number from the JMP color map that corresponds to the *row state*, or a missing value if there is no assigned color.

```
Color Of( Row State() );   // returns color index for current row
Color Of() = 4;            // set current row to Color 4
```

Similarly, `Marker Of` returns or sets the marker index. The marker index is a number from the JMP marker map that corresponds to the active marker, or a missing value if there is no assigned marker.

```
Marker Of();                 // returns marker index for current row
Marker Of( Row State() ) = 4; // set current row to Marker 4
```

Both `Color Of()` and `Marker Of()` accept any row state expression or column or `Row State()` as arguments. They also assume the argument `Row State()` if none is given (some examples are shown with, and some without).

`Color State()` and `Marker State()` are similar to `Color Of()` and `Marker Of()`, except they work in the opposite direction. Where the `-Of` functions turn actual states into indices, the `-State` functions turn indices into states.

```
Row State() = Color State( 4 );  // change current row to green
Row State() = Marker State( 4 ); // change current row to the diamond marker
```

Notice that the last two commands replace the row state with just the color or just the marker, so that any preexisting row state characteristics are lost. More commonly you would issue the `-State` commands for all the characteristics that you want inside a `Combine States()`. The following example puts a green square marker in the third row and hides the row:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Row State( 3 ) = Combine States(
	Color State( 4 ),
	Marker State( 3 ),
	Hidden State( 1 ) );
```

The following script shows the standard JMP markers, which are numbered 0–31. Indices outside the range 0–31 have undefined behavior.

```
New Window( "Markers",
	Graph Box(
		FrameSize( 300, 300 ),
		Y Scale( -1, 16 ),
		X Scale( 0, 7 ),
		For(
			i = 0;
			jj = 15;,
			i < 16;
			jj >= 0;,
			i++;
			jj--;, // 16 rows, 2 columns
			Marker Size( 3 );
			Marker( i, {1, jj + .2} ); // markers 0-15
			Marker( i + 16, {4, jj + .2} ); // markers 16-31
			Text( {1.5, jj}, "marker ", i ); // marker labels 0-15
			Text( {4.5, jj}, "marker ", i + 16 ); // marker labels 16-31
		)
	)
);
```

Figure 9.18 JMP Markers

<hr>

Tip: For more information about this script, see the "Display Trees" chapter on page 517.

<hr>

Note the following about colors:

- JMP colors are numbered 0 through 84.
- The first 16 named colors are the basic colors. See the script below.
- Numbers higher than 16 are darker or lighter shades of the basic colors.
- Indices outside the range 0–84 have undefined behavior.
- For more information about using JMP colors, see "Specify Colors" on page 659 in the "Scripting Graphs" chapter.

The following script illustrates the standard JMP colors:

```
Text Color( 0 );
New Window( "Colors",
    Graph Box(
        FrameSize( 640, 400 ),
        Y Scale( -1, 17 ),
        X Scale( -3, 12 ),
        k = 0;
        For( jj = 1, jj <= 12, jj += 2,
            l = 15;
            For( i = 0, i <= 15 & k < 85, i++,
                thiscolor = Color To RGB( k );
                Fill Color( k );
                thisfill = 1;
                If( thiscolor == {1, 1, 1},
                    Pen Color( 0 );
```

```
                    thisfill = 0;
            ,
                Pen Color( k )
            );
            Rect( jj, l + .5, jj + .5, l, thisfill );
            Text( {jj - 1, l}, "color ", k );
            k++;
            l--;
        );
    );
    jj = -2;
    color = {"Black", "Gray", "White", "Red", "Green", "Blue", "Orange",
"BlueGreen",
        "Purple", "Yellow", "Cyan", "Magenta", "YellowGreen", "BlueCyan",
"Fuschia", "Black"};
    For(
        i = 0;
        l = 15;, i <= 15 & l >= 0,
        i++;
        l--;,
        Text( {jj, l}, color[i + 1] )
    );
) );
```

Figure 9.19 JMP Colors

If you prefer to use RGB values, each color should be a list with percentages for each color in red, green, blue order. For example, the following percentages produce a teal color:

```
Pen Color( {.38,.84,.67} ); // a lovely teal
```

Hue and Shade Example

Hue State and Shade State together are an alternative to Color State for choosing colors. You cannot select black, white, or the shades of gray when you use Hue State. For these, you must use Shade State alone, or Color State.

The following script demonstrates how hue and shade values relate to colors:
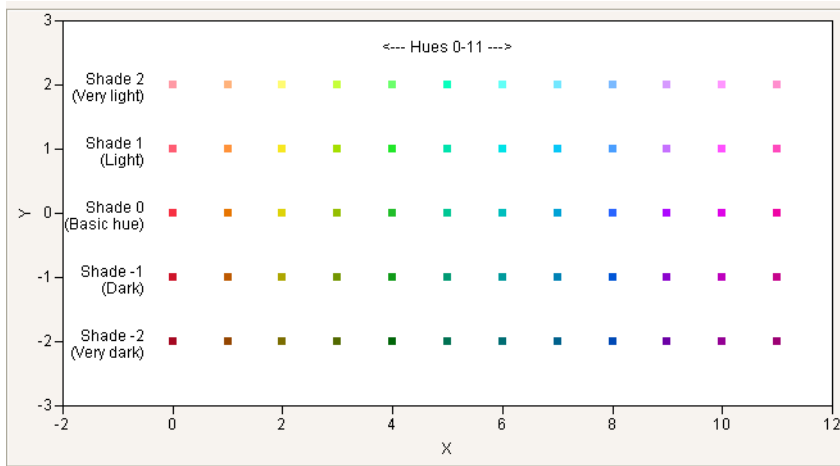
```
New Window( "Hues and Shades",
  Graph Box(
    FrameSize( 600, 300 ),
    Y Scale( -3, 3 ),
    X Scale( -2, 12 ),
    k = 0;
    For( h = 0, h < 12, h++,
      For( s = -2, s < 3, s++,
        myMk = Combine States( Hue State( h ), Shade State( s ), Marker
  State( 15 ) );
        Marker Size( 3 );
        Marker( myMk, {h, s} );
      )
    );
    Text( Center Justified, {5, 2.5}, " <--- Hues 0-11 ---> " );
    Text( Right Justified,
      {-.5, -2}, "Shade -2", {-.5, -2.25}, "(Very dark)",
      {-.5, -1}, "Shade -1", {-.5, -1.25}, "(Dark)",
      {-.5, 0}, "Shade 0", {-.5, -.25}, "(Basic hue)",
      {-.5, 1}, "Shade 1", {-.5, .75}, "(Light)",
      {-.5, 2}, "Shade 2", {-.5, 1.75}, "(Very light)"
    );
  )
);
```

Figure 9.20 Hues and Shades

There are no -Of functions for Hue and Shade. Color Of returns the equivalent Color State index for a color row state that has been set with Hue State or Shade State. For example, the following example gives rows 4 and 5 the same dark red marker:

```
Row State( 4 ) = Combine States( Hue State( 0 ), Shade State( -1 ), Marker
    State( 12 ) );
Row State( 5 ) = Combine States(
    Color State( Color Of( Row State( 4 ) ) ),
    Marker State( Marker Of( Row State( 4 ) ) )
);
```

Row State and Matrices Example

In the following example, row state values are prepared ahead and passed to the Marker routine, along with matrices of coordinates.

```
dt = New Table( "Artificial CP and CA data",
    Add Rows( 26 ),
    New Column( "cover_cp",
        Numeric,
        "Continuous",
        Formula( Random Uniform() / 100 + 0.94 )
    ),
    New Column( "cover_ca",
        Numeric,
        "Continuous",
        Formula( Random Uniform() * 0.04 + 0.94 )
    ),
    New Column( "p", Numeric, "Continuous", Formula( Random Uniform() ) ) )
);
```

```
    dt << Run Formulas;
    greenMark = Combine States( Marker State( 2 ), Color State( 4 ) );
    redDiamond = Combine States( Marker State( 3 ), Color State( 3 ) );
    New Window( "CP and CA Comparisons",
      Graph Box(
        Title( "CP and CA Comparison" ),
        FrameSize( 400, 350 ),
        X Scale( 0, 1 ),
        Y Scale( 0.94, 1 ),
        For Each Row(
          Marker( greenMark, {p, cover_cp} );
          Marker( redDiamond, {p, cover_ca} );
        )
      )
    );
```

Figure 9.21 Example of Row States Values and Matrices

Accessing Data Values

The typical way to work with values in a data table is to follow these steps:

1. Set up the data table whose values you want to access as the current data table. Or, if you already have a data table reference, you can simply use that reference. See "Set the Current Data Table" on page 355.

2. Specify the row or rows whose values you want to access and specify the column name that contains the values that you want to access. See "Set or Get Values by Column Name" on page 458.

The following example opens the Big Class.jmp sample data table (making it the current data table), and then specifies row 2 in the weight column. A value of 123 is returned in the log, which is the weight for Louise in row 2.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:weight[2];
    123
```

Set or Get Values by Column Name

The easiest way to refer to a column is by its name. If you have a global variable and a column with the same name, to prevent ambiguity, scope the column name with the : prefix operator.

To set the value of a cell in the current row, provide the column name and the new value.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Row() = 5; // select row 5
dt:age = 19; // set the age in row 5 to 19
dt:name = "Sam"; // set the name in row 5 to Sam
```

To set the value of a cell in row 10, specify the row number in a subscript.

```
dt:age[10] = 20; // set the value of age in row 10 to 20
```

An empty subscript refers to the current row, so :age[ ] is the same as :age.

To get the value of a cell in row 16, specify the column name.

```
Row() = 16; // select row 16
myGlobal = :age; // store the age as a variable
:age; // returns 14, the age in row 16
Show( :age ); // returns age = 14;
```

To get the value of a cell in a specific row, include the column name and row number. Both of the following examples return 13, the value of age in row 12 of Big Class.jmp:

```
:age[12];
myGlobal = :age[12];
```

To get a value in a data column reference, use Column() and As Column() to get the value in a data column reference. See "Access Cell Values through Column References" on page 395.

Notes:

• If you do not specify the row number, the current row is selected.

• An empty subscript, such as :age[ ], refers to the current row.

- Be careful that you are subscripting to a table row that exists. The default row number is zero, so statements like `:name` that refer to row zero generate an `Invalid Row Number` error.

Additional Ways to Access Data Values

There are other ways to specify a data table, row, and column. You can specify all three items in one expression by using an infix operator and a subscript:

```
dt:age[2] = 12; // table, column, and row
```

If you want to target multiple rows, you can use subscripts with a list or matrix of row numbers.

```
age[i] = 3;
age[{3, 12, 32}] = 14;
list = age[{3, 12, 32}]; // results in a list
vector = age[1 :: 20]; // results in a matrix
dt[1,1] = dt[2,1]; // in Big Class.jmp, "KATIE" (row 1, column 1) is now
  "LOUISE" (from row 2, column 1)
dt[0,{height}] = dt[0,{weight}]; // everybody's height is their weight
dt[1,{height,weight}] = dt[2,4::5]; // row 1 height and weight is from Louise
dt[[5 3 1], 0] = .; /* set rows 5, 3, and 1 to missing. Using out-of-order
  subscripts makes more sense if the right hand side is a matrix or a data
  table with subscripts */
```

Note about Changing Values

Whenever you change values in a data table, messages are sent to the displays to keep them up-to-date. However, if you have thousands of changes in a script, this increases the time it takes to complete the updates.

In order to speed up changes, use `Begin Data Update` before the changes to block these update messages. Use `End Data Update` after the changes have been completed to release the messages and update the displays.

```
dt << Begin Data Update;
...<many changes>...
dt << End Data Update;
```

Be sure to always send the `End Data Update` message, otherwise the display is not updated until forced to do so in some other way.

Note: `Begin Data Update` does not affect the data refresh due to some other table manipulations. For example, when you delete or add columns, the data table is updated and then the data update begins.

Add Metadata to a Data Table

Data tables store observation data, or measurements of various variables on a specific set of subjects. However, JMP data tables can also store *metadata*, or data about the data.

- "Data Table Variables"
- "Table Scripts"
- "Formulas"
- "Delete Metadata"

Data Table Variables

Table variables are for storing a single text string value, such as "Notes". To understand how variables work, first get its existing value by sending a Get Table Variable message:

```
dt = Open( "$SAMPLE_DATA/Solubility.jmp" );
dt << Get Table Variable( "Notes" );
   "Chemical compounds were measured for solubility in different solvents. This
      table shows the log of the partition coefficient (logP) of 72 organic
      solutes in 6 aqueous/nonpolar systems."
```

Now change the existing value of the string using Set Table Variable and then use Get Table Variable again to check that the string has been updated:

```
dt << Set Table Variable( "Notes", "Solubility of chemical compounds" );
dt << Get Table Variable( "Notes" );
   "Solubility of chemical compounds"
```

The following example adds two new table variables to a data table:

```
dt = Open( "$DOCUMENTS/Big Class.jmp" );
myvar = "This is my version of the JMP Big Class sample data.";
dt << Set Table Variable( "key1", myvar );
dt << Set Table Variable( "key2", myvar );
```

Notice that setting the value creates a new variable only if one by the given name does not already exist. If you add two table variables with the same name, only one variable is created.

Table Scripts

You can add a new script, run a script, or get a script using JSL.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Script( "Bivariate Example", Bivariate( Y( :weight), X( :height ) )
   );
dt << Get Table Variable( "Bivariate Example" );
```

```
// returns Bivariate( Y( weight ), X( height ) )
dt << Run Script( "Bivariate Example" )
```

The following example replaces the contents of the table script that was created by the preceding script:

```
dt << Set Property( "Bivariate Example", Bivariate( Y( :weight ), X( :height
    ), Fit Line ) ); // include Fit Line in the Bivariate Example script
```

Suppose you want a text representation of a data table, perhaps to e-mail to a colleague or to use as part of a script. You can obtain a script that reconstructs the information in a data table with `Get Script`. The following example opens Big Class.jmp and prints the data, table variables, and column properties to the log. A portion of the output is shown here:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Get Script;
  New Table( "Big Class",
    Add Rows( 40 ),
    New Script(
      ["en" => "Distribution",...],
      Distribution(
        Continuous Distribution( Column( :weight ) ),
        Nominal Distribution( Column( :age ) )
      ) ),
```

`Delete Scripts` deletes one or more scripts.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Delete Scripts( "Distribution" );
```

In JMP versions prior to 14, use `Delete Property` to delete a script.

Note: The `Names Default to Here` setting is false by default for table scripts. If a table script specifies `Names Default to Here(1)`, this setting lasts only as long as the table script runs. After the script finishes running, the setting is reset to whatever it was before the script ran.

Options for Downloaded Data Tables that Contain Scripts

You are prompted to open or examine downloaded data tables that contain scripts. This help you prevent potentially damaging scripts from running on your computer. In the `Open()` command, you can control this behavior.

To open the data table and enable scripts to run:

```
Open( "$DOWNLOADS/file.jmp", Quarantine Action( "Allow Scripts" ) );
```

To open the data table and block scripts:

```
Open( "$DOWNLOADS/file.jmp", Quarantine Action( "Block Scripts" ) );
```

To display the default prompt window to open or examine the file:

```
Open( "$DOWNLOADS/file.jmp", Quarantine Action( "Show Dialog" ) )
```

To enable a `Try()` expression to capture the error:

```
Try( Open( "$DOWNLOADS/file.jmp", Quarantine Action( "Do Not Open" ) ),
Show( exception_msg ) );
```

If the user selects Examine, the scripts are disabled.

Notes:

• The Block Scripts and Show Dialog options don't work if the domain that you downloaded the data table from is on your Windows Internet Options Trusted sites list.

• After you select Open for a quarantined data table, the data table is trusted until the next time you open JMP.

Prompting to Run a Script upon Opening a Data Table

You can prompt to open a table script named `On Open` or `OnOpen` when the data table opens. The choice is remembered each time you open the data table in the current JMP session.

To create an `On Open` script, perform one of the following actions:

• Create the script using the **Save Script > To Data Table** option, double-click the script name, and change the name to `On Open`.

• Store the script using a `New Script` message.

In this example, you create an OnOpen script in Big Class.jmp.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Script(
  "OnOpen", // create the script
  sortedDt = Data Table( "Big Class" ) << Sort( By( :name ), Output Table
  Name( "Sorted Big Class" ) ) ) // sort the data and put it in a new data table
);
```

The JMP preference called Evaluate OnOpen Scripts determines when the script runs. By default, the user is prompted to run the script. You can set the preference to always run the `On Open` script or to prevent it from running:

```
Preference( Evaluate OnOpen Scripts( "always" ) );
Preference( Evaluate OnOpen Scripts( "never" ) );
Preference( Evaluate OnOpen Scripts( "prompt" ) ); // default setting
```

`On Open` scripts that execute other programs are never run. As a safety precaution, you might consider suppressing automatic execution when opening data tables that you receive from others.

Notes:

- If the OnOpen script is in a downloaded data table, you must specify `Quarantine Action( "Allow Scripts" )` in the `Open()` expression to run the OnOpen script automatically. See "Options for Downloaded Data Tables that Contain Scripts" on page 461.

- When you create a new data table in a script and include the `On Open()` function, `On Open()` is called after the data table is created.

- Some operations in an OnOpen script are considered unsafe, such as saving the current data table. Run an external script to save that table instead of opening a table to run the OnOpen script.

Work with Groups of Table Scripts

Suppose that you want to put related table scripts in a group to save room in the list of data table scripts. Use `Group Scripts`, specify the group name, and then specify the script names to put in the group.

```
dt = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
dt << Group Scripts(
	"Maps",
	{"Bubble Plot Street Map", "Graph Builder Street Map: Traffic Incidents",
	"Graph Builder Street Map Zoomed"}
);
dt << Group Scripts(
	"Tabulate",
	{"Tabulate: Category and Description", "Tabulate: Category, Summary,
	Percentage",
	"Tabulate: Category and Resolution", "Tabulate: Resolution and
	Description"}
);
```

Move a Script Group

`Move Script Group` rearranges the table script groups using the `to first`, `to last`, `after(script)`, and `after(group)` arguments:

```
dt << Move Script Group( "Tabulate", after( "Maps" ) );
dt << Move Script Group( "Tabulate", to last);
```

Get Groups and Group Names

To get the list of table scripts in the group, use `Get Script Group`. Then you can perform an operation such as running a script in the group.

```
tb = dt << Get Script Group( "Tabulate" );
dt << Run Script( tb[2] ); // runs the second script in the Tabulate group
```

`Get Script Groups Names` returns the group names in a list:

```
gn = dt << Get Script Groups Names; // returns {"Maps", "Tabulate"}

// move the Maps group to the end of the table scripts
dt << Move Script Group(gn[1], to last);
```

Select a Script Group

`Select Script Group` selects the named table script group, a list of scripts in a group, or all scripts.

```
dt << Select Script Group; // select all scripts groups
dt << Select Script Group( "Tabulate" ); // select Tabulate group
dt << Select Script Group( {"Tabulate", "Maps"} );// select both groups
```

Ungroup Scripts

`Ungroup Scripts` removes specified table scripts or group from the group. You can send the message to the name of the script group or a list of scripts.

```
dt << Ungroup Scripts( "Tabulate" );
dt << Ungroup Scripts(
    {"Tabulate: Category and Description", "Tabulate: Category, Summary,
    Percentage",
    "Tabulate: Category and Resolution", "Tabulate: Resolution and
    Description"}
);
```

Rename a Script Group

You can rename a table script group using `Rename Script Group`, specifying the existing group name, and specifying the new group name.

```
dt << Rename Script Group( "Maps", "Street Maps" );
```

Delete a Script Groups

To delete a table script group, you get it first and then delete it.

```
a = dt << Get Script Group( "Street Maps" );
dt << Delete Scripts( a );
```

Formulas

The message `Suppress Formula Eval` takes a Boolean argument to specify whether formula evaluation should be suppressed or not. You might want to suppress evaluation if you plan to make numerous changes to the data table and do not want to wait for formula updates between steps.

```
dt << Suppress Formula Eval( 1 );
dt << Suppress Formula Eval( 0 ) ;
```

To accomplish the same effect for all data tables, use the `Suppress Formula Eval` command to turn off formulas globally. This is the same as the message above, except that you do not send it to a data table object.

```
Suppress Formula Eval( 1 ); // make formulas static globally
Suppress Formula Eval( 0 ); // make formulas dynamic globally
```

Note that formulas are not evaluated when they are installed in the column. Even when you force evaluation, they end up being evaluated again in the background. This can be a problem for scripts if they depend on the column having the values while the script is running. If you need a mechanism to control evaluation, use the `EvalFormula` command or the `Run Formulas` command.

To force a single column to evaluate, send an `Eval Formula` command to the column. You can even do this inside the command to create the column, after the formula clause:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Ratio",
    Numeric,
    Formula( :height / :weight ),
    Eval Formula
);
```

`dt << Run Formulas` performs all pending formula evaluations, including evaluations that are pending as a result of evaluating other formulas. This function is useful when you have a whole series of columns to run.

Tip: This method is preferred over `EvalFormula`. Although `EvalFormula` evaluates the formulas, it does not suppress the background task from evaluating them again. The background task takes great care to evaluate the formulas in the right order.

If you send the `Run Formulas` command to a data column, the evaluation is done at the time of the command, but it does not suppress the scheduled evaluations that are pending. Therefore, formulas might end up being evaluated twice if you also send the command to the data table and the data column. Being evaluated twice might be desirable for formulas that have random function in them, or it might be undesirable if they depend on randomization seeds being set. If you use random numbers and use the `Random Reset(seed)` feature to make a replicable sequence, then use the `Run Formulas` command, because it avoids a second evaluation.

Note: All platforms send a `Run Formulas` command to the data table to assure that all formulas have finished evaluating before analyses start.

Set Values without a Formula

`col << Set Each Value(expression)` evaluates the expression for each row of the data table and assigns the result to the column. It does not store the expression as a formula.

Delete Metadata

You can delete table variables, table properties (such as a script or a variable), and formulas from the data table, using the following commands:

```
dt << Delete Table Variable( name );
dt << Delete Table Property( name|{property1, property2, ...});
col << Delete Formula;
col << Delete Property( name|{property1, property2, ...} );
col << Delete Column Property( name );
```

Calculations

This section discusses functions for pre-evaluated columnwise and rowwise statistics and shows how JSL expressions work behind the scenes in the JMP formula calculator.

Pre-Evaluated Statistics

The following functions are special, pre-evaluated functions: `Col Maximum`, `Col Mean`, `Col Minimum`, `Col N Missing`, `Col Number`, `Col Quantile`, `CV` (Coefficient of Variation), `Col Standardize`, `Col Std Dev`, `Col Sum`, `Maximum`, `Mean`, `Minimum`, `NMissing`, `Number`, `Std Dev`, and `Sum`.

Note: Statistics are also computed with `Summarize` ("Store Summary Statistics in Global Variables" on page 364). Although the named arguments to `Summarize` have the same names as these pre-evaluated statistic functions, they are *not* calling the pre-evaluated statistic functions. The resemblance is purely coincidental.

All the statistics are *pre-evaluated*. That is, JMP calculates them once over the rows or columns specified and thereafter uses the results as constants. Because they are computed once and then used over and over again, they are more efficient to use in calculations than the equivalent formula-calculated results.

When JMP encounters a pre-evaluated function in a script, it immediately evaluates the function and then uses the result as a constant thereafter. Therefore, pre-evaluated functions enable you to use columnwise results for rowwise calculations. For example, if you use `Col Mean` inside a column formula, it first evaluates the mean for the column specified and then uses that result as a constant in evaluating the rest of the formula on each row. A formula might standardize a column using its pre-evaluated mean and standard deviation:

```
( Height-Col Mean( Height ) ) / Col Std Dev( Height )
```

For the Big Class.jmp data, `Col Mean(Height)` is 62.55 and `Col Std Dev(Height)` is 4.24. So for each row, the formula above would subtract 62.55 from that row's height value and then divide by 4.24.

Note: Pre-evaluated functions disregard the excluded row state, meaning that any excluded rows are included in calculations. For summary statistics that obey row exclusion, use the Distribution platform.

Columnwise Functions

The functions whose names begin with "`Col`" all work *columnwise*, or down the values in the specified column, and return a single number. For example, `Col Mean(height)` finds the mean of the values in all the rows of the column height and returns it as a scalar result. Some examples include the following:

```
Average Student Height = Col Mean( height );
Height Sigma = Col Std Dev( height );
```

With the `Col` functions, column properties such as Missing Value Codes assign data values that produce incorrect calculations. Suppose that the Missing Value Codes column property is assigned to the x1 column to treat "999" as a missing value. Another column includes a formula that calculates the mean. To use the value "999" instead of a missing value to calculate the mean, refer to `Col Stored Value()` in the formula:

```
Mean( Col Stored Value( :x1 ), :x2, :x3 )
```

Rowwise Functions

The functions *without* "`Col`" listed below work *rowwise* across the values in the variables
specified and return a column result. For example, `Mean(height, weight)` finds the mean of
the height and weight for the current row of the data table. The rowwise statistics are valid
only when used in an appropriate data table row context. The following are some possibilities:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

// scalar result for row 7 assigned to JSL global variable
Row() = 7;
::scalar = Mean( height, weight );

// formula column created in data table
dt << New Column( "Scaled Ht-Wt Ratio",
   Formula( Mean( height, weight ) / age )
);

vector = J( 1, 40 ); // create a 1x40 matrix to hold results
For Each Row( vector[Row()] = Mean( height, weight ) ); // fill the vector
```

Rowwise functions can also take vector (column matrix) or list arguments:

```
myMu = Mean( [1 2 3 4] );
mySigma = Std Dev( {1, 2, 3} );
```

Calculator Formulas

You can store formulas in columns that are automatically evaluated to create the values in the
cells of the column. If you open the formula, you get a calculator interface to edit the formula
structurally. However, the formula is implemented with JSL, and you can obtain the text JSL
form of any expression in the calculator by double-clicking it. The text can be edited, and
when it is de-focused, it is compiled back into the structural form.

Note: There is no difference between a formula column created through the calculator
window and one created directly through JSL with commands such as `New Column(...,`
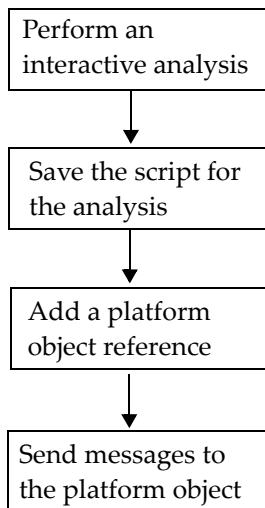`Formula(...))` or `Col << Formula(...)`.

Scripting Platforms

Create, Repeat, and Modify Analyses

If you run the same analysis frequently, you can script it to automate the process. Anyone can then run the script, ensuring the same results every time. To get started, perform your analysis interactively as you normally would with JMP, and then save a script that re-creates that analysis. You can modify the script to further customize it.

Figure 10.1 Typical Workflow for Scripting Platforms

```
┌──────────────────┐
│ Perform an       │
│ interactive      │
│ analysis         │
└──────────────────┘
         │
         ▼
┌──────────────────┐
│ Save the script  │
│ for the analysis │
└──────────────────┘
         │
         ▼
┌──────────────────┐
│ Add a platform   │
│ object reference │
└──────────────────┘
         │
         ▼
┌──────────────────┐
│ Send messages to │
│ the platform     │
│ object           │
└──────────────────┘
```

This chapter is about scripting platforms, not reports. *Platform* object references and *report* object references receive different types of JSL messages. Platforms can run tests, draw plots, and so on. Reports can copy pictures, select display boxes, or close outline nodes. To learn about scripting reports, see the section "Navigate JMP Reports" on page 519 in the "Display Trees" chapter.

Tip: For additional scripting help, see the JMP Scripting Index (**Help > Scripting Index**) and the *JSL Syntax Reference*.

Contents

Example of Scripting a Platform

If you are not familiar with JSL, it can be helpful to start by interactively performing your analysis in JMP, and then saving the analysis to a script. JMP creates the script for you, and you can then add to it or modify it as desired, by creating a platform object and sending it messages.

Overview

This example takes you through these steps:

1. Launch the Analysis and Save the Script to a Window
2. Add a Platform Reference Object and Send Messages
3. Add a Data Table Reference and Save the Script

Launch the Analysis and Save the Script to a Window

1. Select **Help > Sample Data Library** and open Big Class.jmp.

   This fictional data set contains the names, ages, sex, heights, and weights for 40 students.

2. Select **Analyze** > **Fit Y by X**.
3. Select height and click **Y, Response**.
4. Select age and click **X, Factor**.
5. Click **OK**.

   A Oneway plot of students' height by age appears.

6. Click the Oneway Analysis red triangle and select the following options:

   – **Means/Anova**
   – **Compare Means > Each Pair, Student's t**

7. Click the Oneway Analysis red triangle and select **Save Script > To Script Window**.

Here is the resulting script:

```
Oneway(
    Y( :height ), X( :age ),
    Each Pair( 1 ),
    Means( 1 ),
    Mean Diamonds( 1 )
);
```

Notice the following:

• The script begins with a call to the Oneway() function, which returns a Oneway object.

- The parentheses after the `Oneway()` function contain arguments that tell the Oneway function how to make the object.
  - The first two arguments, `Y` and `X`, are required at launch.
  - The next three arguments are optional: `Each Pair`, `Means`, and `Mean Diamonds`.
  - When you selected the Each Pair, Student's t option, the Each Pair feature was turned on.
  - When you selected the Means/Anova option, the Means and Mean Diamonds features were turned on.
- The scripting equivalent of turning an option on or off is the Boolean argument 0 (off) or 1 (on).

Add a Platform Reference Object and Send Messages

1. (Optional) Right-click in the script and select **Show Embedded Log**.

   Output that appears in the log is now easily visible below the script.

2. Choose a JSL variable to remember the Oneway platform object.

   In this example, use the name `oneObj`:

   ```
   oneObj = Oneway(
      Y( :height ),
      X( :age ),
      Each Pair( 1 ),
      Means( 1 ),
      Mean Diamonds( 1 )
   );
   ```

   This gives you a way to address the platform object and send messages to it.

3. Click **Run Script** 📄 .

   This creates the Oneway object and sets the variable `oneObj`. Now, send a message to tell the platform to turn on the Unequal Variances report.

4. At the end of the script, add the following line: `oneObj << Unequal Variances(1)`, as shown here:

   ```
   oneObj = Oneway(
      Y( :height ),
      X( :age ),
      Each Pair( 1 ),
      Means( 1 ),
      Mean Diamonds( 1 )
   );
   oneObj << Unequal Variances( 1 );
   ```

5. Highlight the Unequal Variances line of JSL and click **Run Script** 📄 .

Tip: You can also press Ctrl+R to run a highlighted line of JSL.

At the bottom of the report window, the Tests That the Variances Are Equal report now appears. Now, suppose that you want to see only that report (and the graph), and you want to close the other reports.

6. Add the following lines to the script:

```
rep = Report( oneObj );
rep["Oneway Anova"] << Close( 1 );
rep["Means Comparisons"] << Close( 1 );
```

The Report() function returns the report object for the Oneway platform, and stores a reference to the report in the JSL variable called rep. You can send messages or perform actions on the report object, such as closing specific report outlines.

7. Highlight these last 3 lines of the script and click **Run Script** .

In the report window, the Oneway Anova and the Means Comparisons reports are now closed. Only the initial graph and the Tests that the Variances are Equal report (which includes the Welch's Test report) appear.

If you close the Big Class.jmp sample data table at this point and then try to run the script, you are prompted to open a data table. A best practice is to precede the call to the Oneway platform with an Open() function to open the associated data table each time the script is run.

Add a Data Table Reference and Save the Script

1. Add the following line as the first line in the script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

If the data table is not open when the script is run, this line opens the associated data table.

Your finished script should look like this:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
oneObj = Oneway(
    Y( :height ),
    X( :age ),
    Each Pair( 1 ),
    Means( 1 ),
    Mean Diamonds( 1 )
);
oneObj << Unequal Variances( 1 );
rep = Report( oneObj );
rep["Oneway Anova"] << Close( 1 );
rep["Means Comparisons"] << Close( 1 );
```

Once you complete your script, save it for future access.

2. From the script window, select **File** > **Save As**. Enter a name for your script and save it to any directory.

Send Messages to a Platform

Before you can send messages to a platform, you need to create a platform object that you can send the messages to. Use the Send operator (<<).

The following line keeps a reference to a Oneway platform in the JSL variable called oneObj:

```
oneObj = Oneway( Y( height ), X( age ) );
```

Then, add the Send operator to send the platform object a message. In this example, the message is to turn on the Unequal Variances report:

```
oneObj << Unequal Variances( 1 );
```

To see which messages can be sent to a specific object, do one of the following:

*   Use the autocomplete shortcut. On a new line, enter the object name, then the Send operator (<<), and press Ctrl+space (Windows) or Ctrl+Option+space (macOS). The following example shows the messages that you can send to the platform object oneObj on Windows:

```
oneObj << (Ctrl+space)
```

A list of messages appears. Click one to insert it.

*   Use the Show Properties () function. Specify the object in the parenthesis. The following example shows the messages that you can send to the platform object oneObj:

```
Show Properties (oneObj)
```

A list of messages appears in the log.

Conventions for Messages and Arguments

*   For most messages, omitting the argument for a Boolean command enables the option. For example, all of these messages create a test for unequal variances:

```
oneObj << Unequal Variances;
oneObj << Unequal Variances( 1 );
oneObj << Unequal Variances( "true" );
oneObj << Unequal Variances( "yes" );
oneObj << Unequal Variances( "present" );
oneObj << Unequal Variances( "on" );
```

The first three approaches are recommended.

Notes:

- The quoted arguments `"present"`, `"absent"`, `"on"`, `"off"`, `"switch"`, and `"flip"` are supported for backwards compatibility.

- When the name of the JMP option contains several options separated by a comma or slash (such as the **Means/Anova** option or the **T Square, T^2** option) you can use any *one* of the menu names. Or, you can use the full menu name, including the comma or slash, if the string is followed by n. For example, any of these messages create a Means/Anova test:

  ```
  oneObj << Means( 1 )
  oneObj << Anova( 1 )
  oneObj << "Means/Anova"n( 1 )
  ```

- When an option appears under a menu in JMP, the corresponding script message is the option itself without the parent menu. For example, in the Oneway red triangle menu, the Nonparametric menu has multiple options under it, including Wilcoxon Test. Simply use Wilcoxon Test as a function or message in a script:

  ```
  oneObj = Oneway( Y( height ), X( age ), Wilcoxon Test( 1 ));
  oneObj << Wilcoxon Test( 1 );
  ```

- When a menu in JMP contains values rather than options, in the script, specify the parent menu and the value as an argument. For example, in the Oneway red triangle menu, the Set Alpha Level menu has values, such as 0.10, 0.05, 0.01, and Other. To specify 0.01 in a script, add a line like the following:

  ```
  oneObj << SetAlphaLevel(0.01);
  ```

Send Multiple Messages

To send several messages, just add more `<<` operators or more `Send` arguments:

```
dist << Quantiles( 1 ) << Moments( 1 ) << More Moments( 1 ) << Horizontal
  Layout( 1 );
Send( dist, Quantiles( 1 ), Moments( 1 ), More Moments( 1 ), Horizontal
  Layout(1));
```

Because `<<` is an *eliding operator*, it combines arguments and works differently than if its arguments were grouped. You can stack multiple messages with extra `<<` symbols to perform them all in order, from left to right.

Another way to stack messages is to send a list of messages:

```
dist << {Quantiles( 1 ), Moments( 1 ), More Moments( 1 ), Horizontal
  Layout( 1 )};
```

These approaches work well assuming that no value is returned as a result of one of your messages. However, if your goal is to send a message to the *result* of another message, add grouping parenthesis. The second line shows the wrong way to do it; the third line shows how to do it correctly:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
/* the following line is incorrect, and sends the Fit Mean message to the
   original data table object */
bv = dt << Run Script( "Bivariate" ) << Fit Mean( 1 );
/* the following line is correct, and sends the Fit Mean message to the
   resulting Bivariate object/report */
bv2 = ( dt << Run Script( "Bivariate" ) ) << Fit Mean( 1 );
```

Find Messages for Objects

Some messages apply to all platform objects, and others are specific to certain platforms. Once you have created a platform object, find out what messages you can send to the object, using any of the following approaches:

Use the Scripting Index in JMP

1. Select **Help** > **Scripting Index**.
2. Enter the object type that you are interested in.
3. Click the correct object in the resulting list. Messages for the object appear in the Items list.

Once you have created a platform object, use the Show Properties(obj) **function.**

Replace obj with your named platform object. A list and brief description of all the messages the object can receive appears in the Log. To understand more about the Show Properties output, see "Interpret the Show Properties List" on page 478.

Tip: The Show Properties() function also works with data tables and display boxes. See "How Can I See All of the Messages that Can be Sent to a Data Table Object?" on page 326 in the "Data Tables" chapter and "The << Operator" on page 530 in the "Display Trees" chapter.

• Examine the platform in JMP launch windows and report windows. Most options that you find there have JSL equivalents with the same names and arguments.

Interpret the Show Properties List

Show Properties lists the messages that can be sent to a scriptable object, such as a data table, an analysis platform, or a display, producing a text report in the log. Each type of message is categorized and appears in brackets, and can be any of the following:

Message Type	Description	Example
[Action]	Correspond to options in the JMP interface.	Redo Analysis [Action] (Rerun this same analysis in a new window. The analysis will be different if the data has changed.)
[Action Choice] [Enum]	List specific choices for their arguments.	Fit Polynomial [Action Choice] {2, quadratic, 3, cubic, 4, quartic, 5, 6} Draw [Enum] {Filled, Outlined, Filled and Outlined}
[Boolean]	Turn options on or off. Arguments are usually 1 or 0. If a message is specified without an argument, sending the message changes it to the opposite state. [Default On] indicates that the option is on by default.	Show Points [Boolean] [Default On]
[Subtable]	List options within submenus. You use the option itself, not the parent item.	Script [Subtable] Redo Analysis [Action] (Rerun this same analysis in a new window. The analysis will be different if the data has changed.) Save Script to Datatable [Action] (Return to the launcher for this analysis.)
[New Entity]	Open a new window in the interface.	Prediction Interval [New Entity] (Prediction Interval to contain a single future observation or the mean of m future observations.)

For example, run the following JSL:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate( Y( :height ), X( :age ) );
Show Properties( biv );
```

The object biv supports Subtable, Boolean, Action, and Action Choice type messages.
Enum and New Entity messages are not supported.

Send Messages to an Object

Suppose that you want to send a message to an object in the output. For example, in Fit Model,
you want to send the Response Limits message to the Profiler object, Use the subscript
operator ([ ]) to refer to the Profiler object.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

fm = dt << Fit Model(
  Y( :height ),
  Effects( :age, :sex, :weight, :age * :weight, :sex * :weight ),
  Personality( Standard Least Squares ),
  Emphasis( Effect Screening ),
  Run
);

obj = fm << Profiler(
  1,
  Confidence Intervals( 1 ),
  Desirability Functions( 1 ),
  Term Value( age( 12 ), sex( "F" ), weight( 105 ) )
);

Show( Profiler[1] << Get Desirability ); // show the limits before

Profiler[1] << (height << Response Limits(
  {Lower( 50, 0.95 ), Middle( 58, 0.7 ), Upper( 70, 0.066 ),
  Goal( "Minimize" ), Importance( 1 )}
));
Wait( 0.1 ); // for demonstration purposes
Show( Profiler[1] << Get Desirability ); // show the limits after
```

Specify Which Columns to Analyze

If you created your script interactively, you already specified columns in the launch window, so there is no need to specify them again. However, when creating a script from scratch, you normally want to specify the columns to analyze. For example, the following line launches the Distribution platform and specifies the height and weight columns as the Y variables:

```
Distribution( Y( height,weight ));
```

Create Column References

Tip: To see all the messages you can send to a column, go to **Help** > **Scripting Index** > **Data Table** > **Column Scripting**.

Creating a column reference enables you to access and send messages to the column. If you store each column reference in a JSL variable, you can use the variables instead of column names in the Y and X roles and elsewhere in the script.

For example, using Big Class.jmp, the following lines create Ycol as a column reference for weight, and Xcol as a column reference for height. The last line sends a message to Xcol to get the data type.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Ycol = Column( "weight" );
Xcol = Column( "height" );
Xcol << Get Data Type;
```

In the log, "Numeric" is output for the data type of the Xcol (height column).

Referring to a column reference with the **:** scoping operator (rather than plain text) is convenient in scripts saved to a data table. If the columns are renamed in the data table, the column references in the script are updated.

Specify Multiple Column Names at Once

If you have a lot of column names, instead of listing each one individually, you can use lists as column arguments. The following example gets the Y column names from a list and creates a Oneway analysis on each column in the list.

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Fit Group( Oneway( Y( {:SO2, :POP} ), X( :State ) ) );
```

If you do not know the column names when you are writing the script, use column indices, as done in the following example.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Distribution( Y( 1::N Col( dt ) ) );
```

You can also use indices when specifying effects as shown here:

```
dt << Fit Model( Y( :y ), Effects( 1 :: 4 ) );
```

Here's another example of specifying multiple column names. 5::31 selects columns 5 through 31.

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA/Baltic.jmp" );
dt << Partial Least Squares( Y( :ls, :ha, :dt ), X( 5 :: 31 ), Go );
```

Enable Users to Specify Columns

Instead of specifying analysis columns yourself, you can enable a user to specify them, but you can still control what happens after they do so. In this case, use an empty specification. The following example runs the Distribution launch window:

```
Distribution();
```

In this scenario, the Distribution launch window appears and the user can choose the columns to analyze. After the user selects the columns and clicks **OK**, the Distribution report is shown as specified.

Specify a By Variable

In many JMP platforms, you can specify columns as By variables. To do this in a script, include a By argument in the platform command, listing each column as an argument.

The following example uses the Big Class.jmp data table, which contains the names, ages, sex, heights, and weights for 40 students. Create a bivariate report of weight by height, using sex as a By variable, and adding a variety of fits.

Create the Bivariate Report Using a By Variable

1. Select **File** > **New** > **Script**.
2. Add the following lines:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

```
biv = dt << Bivariate( Y( weight ), X( height ), By( sex ));
```

The first line opens the Big Class.jmp sample data table. The second line creates a platform
object called biv that runs a bivariate report of weight and height by sex.

3. Click **Run Script** 📄 .

The report window shows two graphs, one where sex is F and one where sex is M.

Send Messages to the Entire Platform or to a Single By Level

1. If you do not have the log showing, turn it on. Right-click in the script and select **Show
   Embedded Log**.

2. Add this line to the script:

   ```
   Show( biv );
   ```

3. Highlight the Show(biv) line and click **Run Script** 📄 .

   In the log, rather than returning a single reference to a platform, Bivariate[], the
   platform object returns a list of references: biv = {Bivariate[], Bivariate[]}. The
   two references correspond to the two levels (F and M) of the By variable, sex.

   You can direct messages to each By level individually or to all By levels.

4. Send a message to all By levels to add a linear regression fit. Add this line to the script:

   ```
   biv << Fit Line;
   ```

5. Highlight the line you just created and click **Run Script** 📄 .

   In the report window, a linear regression line is added to both graphs, with the
   corresponding Linear Fit reports.

6. Send a message to only the F level, adding a cubic polynomial fit. Add this line to the
   script:

   ```
   biv[1] << Fit Polynomial( 3 );
   ```

   Tip: The number for each By level corresponds to their order in the report window, which
   is usually alphanumeric. If the By column contains a Value Order column property, that
   ordering is followed.

7. Highlight the line you just created and click **Run Script** 📄 .

   In the report window, a polynomial fit line is added to the F graph, with the corresponding
   Polynomial Fit reports.

8. Send a message only to the M level, adding a quartic polynomial fit. Add this line to the
   script:

   ```
   biv[2] << Fit Polynomial( 4 );
   ```

9. Highlight the line you just created and click **Run Script** 📄 .

In the report window, a quartic fit line is added to the M graph, with the corresponding Polynomial Fit reports.

Figure 10.2 By Group Reports

If you specify more than one column in `By` argument, graphs appear for each subgroup of each By variable. In this example, `By( sex, age )` would produce graphs for females age 12, females age 13, and so on, up to age 17. It would also produce graphs for males age 12 up to age 17.

Extract Results from a Report

The following example shows how to launch a platform with By groups and extract results (in this case, variances) from each group:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
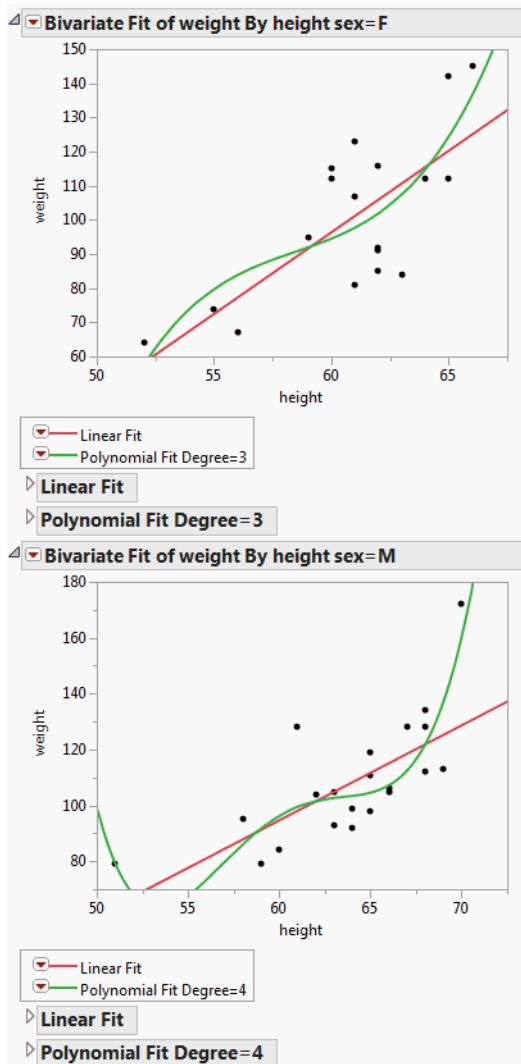
// onew is the JSL variable holding a list of platforms
onew = dt << Oneway( x( :age ), y( :height ), by( :sex ), anova );

// r is a list of reports
r = onew << Report;

// nBy is the number of reports generated
nBy = N Items( r );

// vc is an array of as many rows as reports, one column, all set to zero
vc = J( nBy, 1, 0 );

// for each report, do the following:
For( i = 1, i <= nBy, i++,
   vc[i] = r[i] // vc[i] is the variance of the ith report
[Outline Box( "Analysis of Variance" ), // look for this outline

// then look for this column and get the second value
   Column Box( "Sum of Squares" )][2]
);
Show( vc ); // debugging, look in log to see this value

// byValues becomes a list of the values in the sex column
Summarize( byValues = By( :sex ) );

// make a new table with two rows (M.F) and two columns
New Table( "Variances" )
<< New Column( "Sex", // create a new column called Sex
   character,
   width( 8 ),
   values( byValues )

// create a new column called Variance
) << New Column( "Variance", Numeric, "Continuous", Values( vc ) );
```

Filter by Value using a Where Statement

You can use a `Where` statement to filter by a specific value. If you are adding a Where clause to a script that contains column references, you must first resolve the references. The following example limits the analysis to females and fits a line on the Bivariate report.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Ycol = Column( "weight" );
Xcol = Column( "height" );
biv2 = dt << Bivariate(
Y( Ycol ),
X( Xcol ),
Fit Line( ),
Where( :sex == "F" ) );
```

Enable User Input

When you send an `Action()` function to a platform, it evaluates an expression. For example, use `Action` messages when you want the launch window to appear. A user chooses columns in the launch window and the script continues running after the user runs the analysis.

The following example launches four platforms: Distribution, Bivariate, Oneway, and Contingency. The launch window for each platform appears, and the user chooses the analysis columns and clicks OK. The report window for each analysis appears, and then the next platform launch window comes up, until all platforms have been launched.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Distribution( Action( doit ) );
doit = Expr(
  New Window( "Bivariate", Bivariate( Action( doit2 ) ) )
);
doit2 = Expr(
  New Window( "Oneway", Oneway( Action( doit3 ) ) )
);
doit3 = Expr(
  New Window( "Contingency", Contingency( Action( doit4 ) ) )
);
doit4 = Expr();
```

Execute Embedded Red Triangle Options

In addition to the red triangle menu at the top of a report, some report windows contain embedded red triangle menus. These appear when you select an option that has sub-options, such as profilers. To execute options in embedded red triangle menus, use the Get Scriptable Object message. The following example creates a Life Distribution report then turns off the Confidence Intervals option in the Hazard Profiler red triangle menu:

```
dt = Open( "$SAMPLE_DATA/Reliability/Fan.jmp" );

// create a Life Distribution report with a lognormal distribution
obj = dt << Life Distribution(
        Y( :Time ),
        Censor( :Censor ),
        << Fit Lognormal
);

// get a reference to the Hazard Profiler red triangle menu
haz_prof = ( report( obj )[ "Hazard Profiler" ] << Get Scriptable Object);

// turn off confidence intervals in the Hazard Profiler
haz_prof << Confidence Intervals( 0 );
```

Make Platforms Invisible

In certain circumstances, you might want to make the platform analysis invisible. This means that no platform windows appear, and the platform runs in the background.

If you are doing a simulation or bootstrap analysis in your script, the script might call a platform hundreds or thousands of times. Perhaps you just want to pull one or two results out of each platform report. The following example performs a simulation and then creates a table of results and shows them in a Distribution report:

```
dt = As Table( (0 :: 10)`, << Column Names( {"X"} ) ); // create X variable

// set initial random seed to get reproducible results
Random Reset( 12345 );
dt << New Column( "Y",
   "Continuous",
   "Numeric",
   Set Formula( :X * 2 + 1 + Random Normal() )
);
```

```
// Y values have a random component
res = []; // define results matrix (empty to start)
For( i = 1, i <= 100, i++, // loop through simulation runs

  // for each iteration, fit a regression line and save parameter estimates
  bv = dt << Bivariate( Y( :Y ), X( :X ), Fit Line( 1 ), Invisible );
  res |/= (Report( bv )["Parameter Estimates"][

  // concatenate the results to the bottom of the results matrix
  Number Col Box( "Estimate" )] << Get as Matrix)`;

  // close the window (even though it's invisible)
  bv << Close Window;
     dt:Y << Eval Formula; // generate new Y values
);
// create table of results
dtres = As Table( res, << Column Names( {"Intercept", "Slope"} ) );

// show results in Distribution
dtres << Distribution( Y( :Intercept, :Slope ) );
```

If you want to run an analysis as an intermediate step in your script, and do not need the user to see the analysis, hide it with the `invisible` option. In this case, you might want to show only specific results from the report, which you can output to a journal or to the log.

- The following example runs the Distribution platform as invisible, and outputs only the CDF plot to a new journal:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dist = dt << Distribution( Y( :height ), CDF Plot( 1 ), invisible );
Report( dist )["CDF Plot"] << Journal;
dist << Close Window;
```

- The following example extracts the *F*-Ratio from an invisible Bivariate report, and outputs it to the log:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( x( :height ), y( :weight ), invisible );
biv << Fit Line;
r = biv << Report;
fratio = r[ColumnBox( "F Ratio" )][1];
r << Close Window;
Show( fratio );
   fratio = 1.15609545178219;
```

Tip: Invisible windows use resources that must be manually freed. Be sure to close the invisible window when the script is done with it.

You can also use the `invisible` option on the options in the Tables menu, such as Subset, Sort, Stack, and so on. The following example makes the Subset operation invisible:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :age == 14 );
dt = dt << Subset( invisible );
subDt << Bivariate( x( :height ), y( :weight ), Fit Line );
```

Note that the preceding line could also be handled using a `WHERE` clause:

```
subDt << Bivariate( x( :height ),y( :weight ), Where( :age == 14 ), Fit Line
    );
```

Specify Report Titles

You can specify the title (shown in the title bar of a platform's report) by adding the `title` command to the launch request. The following example replaces the standard bivariate report's title with a user-specified one (My Title).

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Bivariate( x( :height ), y( :weight ), Title( "My Title" ) );
```

Common Messages for Platform Windows

For a list of common messages that you can send to most platforms, see the *JSL Syntax Reference*.

Scripting Considerations by Platform

This section explains specific considerations when scripting the following platforms:

- "Categorical" on page 489
- "Control Charts" on page 490
- "Distribution" on page 492
- "DOE" on page 493
- "Fit Model" on page 495
- "Formula Depot" on page 500
- "Graph Builder" on page 500
- "Neural and Neural Net" on page 502

Categorical

The Free Text option in Categorical displays a Text Explorer report. Scripts that call the Free Text option in Categorical should be updated to use the Text Explorer platform.

Scripting for Supercategories

When ratings are involved in a data set (for example, a five point scale), you might want to know the percent of the responses in the top two or another subset of ratings. You can define a group of ratings in the data using the supercategories scripting option (instead of the column property).

The following example creates a new column called Floss Delimited 2, and adds a supercategory called Sleep Related, which contains two values: Before Sleep and Wake.

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
dt << New Column( "Floss Delimited 2",
   Character,
   Set Property(
      "Supercategories",
      {Group( "Sleep Related", {"Before Sleep,", "Wake,"} )}
   ), ...);
```

In the data table, under the Columns panel at left, scroll down to the new Floss Delimited 2 column. Click the column property indicator ✱ and select Supercategories. You can see the new Sleep Related supercategory with its two values.

Supercategories can also be specified in a Categorical launch command, where the properties are listed inside parentheses after the column name:

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
dt << Categorical(
   Supercategories(
      :Floss Delimited(
         {Group( "Sleep Related", {"Before Sleep,", "Wake,"} )}
      )
   ),
   Multiple Response( :Floss Delimited )
);
```

In the Categorical report, notice that the Sleep Related category has 203 cases. If you had run the Categorical platform without creating this supercategory, Before Sleep and Wake would appear as separate categories with 143 and 60 cases, respectively.

For more information about supercategories, see *Consumer Research*.

Control Charts

Using JSL, you can create control charts, customize tests, run alarm scripts, and more.

Customize Tests in Control Chart Builder

You can design custom tests and select or deselect multiple tests at once using the Customize Tests function. You specify the description, desired number, and label. This option is available only for Variables and Attribute chart types.

The following example creates a custom test called Test 1 and turns it on.

```
dt = Open( "$SAMPLE_DATA/Quality Control/Diameter.jmp" );
dt << Control Chart Builder(
  Variables( Subgroup( :DAY ), Y( :DIAMETER ) ),
  Customize Tests( Test 1( 2, "1" ) ), // test number, n, label
  Chart( Position( 1 ), Warnings( Test 1( 1 ) ) ), // turn Test 1 on
  Chart( Position( 2 ) )
);
```

Run Alarm Scripts

An alarm script alerts you when the data fail one or more tests. As an Alarm Script is invoked, the following variables are available, both in the issued script and in subsequent JSL scripts:

qc_col is the name of the column

qc_test is the test that failed

qc_sample is the sample number

qc_phase is the label of the phase during which the failure occurred

qc_firstRow is the first row in the sample

qc_lastRow is the last row in the sample

Example 1: Automatically Writing to a Log

One way to generate automatic alarms is to make a script and store it with the data table as a data table script or column property named QC Alarm Script.

The following example automatically writes a message to the log whenever a test fails:

```
dt = Open( "$SAMPLE_DATA/Quality Control/Coating.jmp" );
obj = dt << Control Chart(
  Sample Size( :Sample ),
  KSigma( 3 ),
  Chart Col( :Weight, XBar, R ),
  Alarm Script(
    Write(
      "Out of Control for test ",
      qc_test,
      " in column ",
      qc_col,
      " in sample ",
      qc_sample
    )
  )
);
obj << Test 1( 1 );
```

Example 2: Running a Chart with Spoken Tests

You can create a control chart with tests that are spoken aloud using the Speak function. Try the following example:

```
dt = Open( "$SAMPLE_DATA/Quality Control/Coating.jmp" );
dt << Control Chart(
  Alarm Script(
    Speak(
      Match( QC_Test,
        1, "One point beyond Zone A",
        QC_Test, 2,
        "Nine points in a row in zone C or beyond", QC_Test,
        5,
          "Two out of three points in a row in Zone A
          or beyond"
      )
    )
  ),
  Sample Size( :Sample ),
  Ksigma( 3 ),
  Chart Col( :Weight, Xbar( Test 1( 1 ), Test 2( 1 ), Test 5( 1 ) ), R )
);
```

You can have either of these scripts use any of the JSL alert commands, such as `Speak`, `Write`, or `Mail`.

Set Phase Limits

A *phase* is a group of consecutive observations in the data table. For example, phases might correspond to time periods during which a new process is brought into production and then put through successive changes. Phases generate, for each level of the specified Phase variable, a new sigma, set of limits, zones, and resulting tests.

The following example illustrates setting the limits for the different phases of Diameter.jmp:

```
dt = Open( "$SAMPLE_DATA/Quality Control/Diameter.jmp" );
dt << Control Chart(
    Phase( :Phase ),
    Sample Size( :DAY ),
    KSigma( 3 ),
    Chart Col(
        :DIAMETER,
        XBar(
            Phase Level(
                "1",
                Sigma( .29 ),
                Avg( 4.3 ),
                LCL( 3.99 ),
                UCL( 4.72 )
            ),
            Phase Level(
                "2",
                Sigma( .21 ),
                Avg( 4.29 ),
                LCL( 4 ),
                UCL( 4.5 )
            )
        ),
        R( Phase Level( "1" ), Phase Level( "2" ) )
    )
);
```

Distribution

You can set specification limits using the Spec Limits column property or using the `Spec Limits` argument in JSL:

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Distribution(
```

```
      Column( :OZONE ),
      Fit Distribution(
         Weibull(
            Spec Limits( LSL( 0.075 ), Target( 0.15 ), USL( 0.25 ) ) )
         )
      )
   );
```

To set specification limits using K Sigma normal probabilities, use the `Set Spec Limits for KSigma` message. The first argument to the message is the value of K Sigma. By default, the limits are two-sided. To make them one-sided, use the optional `sided` argument. Specify `sided=1` for LSL only and `sided=2` for USL only.

```
   dt = Open( "$SAMPLE_DATA/Cities.jmp" );
   obj = Distribution( Column( :CO ) );
   obj << Fit Distribution(
      LogNormal( Set Spec Limits for KSigma( 3 ) )
   );
```

DOE

Tip: For a complete list of DOE scripting options, select **Help > Scripting Index**, select **Objects** from the menu, and search for DOE.

To reproduce your DOE work, most DOE platforms provide these approaches:

- In the red triangle menu for the DOE window, the Save Script to Script Window option provides a script that reproduces the work that you have entered. This option is not available for the Nonlinear Design and Taguchi Array platforms.

- In the output design tables created by most DOE platforms, a script called DOE Dialog reproduces the work in the DOE window that created the design table. This script also contains the random seed (and number of starts or column starts, where appropriate) that enables you to reproduce your design exactly.

For many DOE platforms, the design table that is created also contains one or more scripts that you can run to conduct an appropriate analysis. These scripts are prefilled with the appropriate settings for your design.

The Random Seed

In constructing designs using the DOE platform, JMP uses a random seed to control certain actions that have a random component. A random seed can control the following:

- initializing search algorithms for design generation
- randomizing Run Order for design construction

- selecting a starting design for designs based on random starts

To reproduce a design using a script, you need to specify the random seed that generated them. For designs using random starts, set a random seed in the script before making the design.

The following example creates a Custom Design, sets the random seed, and makes the design:

```
DOE(
    Custom Design,
    Add Factor( Continuous, -1, 1, "X1", 0 ),
    Add Factor( Continuous, -1, 1, "X2", 0 ),
    Set Random Seed( 34067086 ),
    Make Design
);
```

Reproduce a Custom Design or a Main Effects Screening Design

Custom designs and main effects screening designs (constructed in the Screening Design platform) are generated by allocating a maximum number of seconds (Design Search Time) spent searching for a design. The default setting for Design Search Time is based on the complexity of the design. Even when two custom or main effects screening designs are constructed using the same random seed, the designs might not be identical because of different machine processing capabilities.

When you save the script for either of these designs using the Save Script to Script Window option, or when the DOE Dialog script is saved to the data table, the script specifies the Random Seed and the Number of Starts. The Number of Starts is the number of random starts used during the Design Search Time. The combination of the Set Random Seed and the Number of Starts options enables you to reproduce the design.

Note: If you write your own script, be aware that the order in which you specify options in your script can affect your results. For example, you should specify the design options before you make the design. In particular, if your design is constructed using a random seed, set it before you make the design.

Global Variables

Here are some global variables used to initiate or tune search algorithms.

Starting Design   Enables you to specify a starting design. The following example replaces the random starting design with a specified matrix:

```
DOE Starting Design = matrix;
```

If a starting design is supplied, the Custom Design platform has only one start using this design.

K Exchange Value   The coordinate exchange algorithm considers every row of factor settings for possible replacement in every iteration. You can limit the number of rows considered for exchange in an iteration so that only a small number of most likely rows is considered. The following example tells the algorithm to consider only the three most likely rows for exchange in each iteration:

```
DOE K Exchange Value = 3;
```

Bayes Diagonal   Defines a vector that is added to the diagonal elements of the $X'X$ matrix. This new matrix is used in finding the D-optimal design. The following example adds the elements of vector to the diagonal elements of the $X'X$ matrix:

```
Bayes Diagonal = vector:
```

Fit Model

When scripting the Fit Model platform, here are a few general tips for controlling the behavior of the launch window:

- To keep the launch window open and fit the model at the same time, do one of the following (they are equivalent):
  - Include the Run Model message in your script.
  - Include both the Run message and the Keep Dialog Open(1) message.
- To run the model without showing the launch window, include the Run message in your script.

Model Scripts in the Data Table

When a data table contains a *Model* script and you launch Fit Model interactively, the Fit Model launch window is pre-populated according to the roles assigned by the table script.

Fit Group

If you want to fit different models and see a shared Profiler so that you can compare the models, use the Fit Group function. You can fit models such as least squares, nonlinear, neural, Gaussian processing, and mixed, in the same window with a shared Profiler. The following example creates both a Standard Least Squares model and a Gaussian Process model.

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
obj = dt << Fit Group(
  Fit Model(
    Y( :ABRASION ),
    Effects(
      :SILICA & RS,
```

```
            :SILANE & RS,
            :SULFUR & RS,
            :SILICA * :SILICA,
            :SILICA * :SILANE,
            :SILANE * :SILANE,
            :SILICA * :SULFUR,
            :SILANE * :SULFUR,
            :SULFUR * :SULFUR
        ),
        Personality( "Standard Least Squares" ),
        Emphasis( Minimal Report ),
        Run()
    ),
    Gaussian Process(
        Y( :HARDNESS ),
        X( :SILICA, :SILANE, :SULFUR ),
        Set Correlation Function( "Cubic" )
    )
);
```

In the report, click the Fit Group red triangle and select **Profiler**. Scroll down to the Prediction
Profiler. You can see the ABRASION row for the SLS model, and the HARDNESS row for the
Gaussian Process model appearing in the same profiler.

Effects

Effects in the model can be more than just a list of columns, and can have a specialized syntax:

```
    Effects( list of effects, list of effect macros, or both lists );
```

An effect can be a column name, a crossing of several column names with asterisk (*) notation,
or nested columns specified with subscript bracket ([ ]) notation. Additional effect options
can appear after an ampersand (&) character. Here are some examples:

```
    A,                   // a column name alone is a main effect
    A*B,                 // a crossed effect, interaction, or polynomial
    A[B],                // nested
    A*B[C D],            // crossed and nested
    effect&Random,       // a random effect
    effect&LogVariance,  // a variance model term
    effect&RS,           // a response surface term
    effect&Mixture,      // for an effect participating in a mixture
    effect&Excluded,     // for an effect that with no model arguments
    effect&Knotted,      // for a knotted spline effect
```

Effect macros are:

```
    Factorial( columns ),   // for a full factorial design
    Factorial2( columns ),  // for up to 2nd-degree interactions only
```

```
Polynomial( columns ),  // for a 2nd-degree polynomial only
```

Responses and Effects for MANOVA

To address an individual response function analysis, use a subscripted `Response`:

```
manovaObj << ( Response[ 1 ] << {response options} );
manovaObj << ( Response[ "Contrast" ] << {response options} );
```

Each response function supports the `Custom Test` message:

```
Custom Test( matrix, <Power Analysis( ... )>, <Label( "..." )> )
```

where each row of the *matrix* specifies coefficients for all the arguments in the model.

To address an individual Effect test, use subscripted `Effect` with a name or number:

```
manovaObj << ( Response[1] << ( Effect["Whole Model"] << {effect options} ) );
manovaObj << ( Response[1] << ( Effect[i] << {effect options} ) );
```

The effects are numbered:

- 0 for the intercept
- 1, 2, and so on, for regular effects
- *n*+1 for the "Whole Model" test, where *n* is the number of effects *not* including the intercept

Each effect in each response function supports the following messages, where each row of the *matrix* has coefficients for all the levels in the effect:

```
Test Details( 1 ),
Centroid Plot( 1 ),
Save Canonical Scores,
Contrast( matrix, <Power Analysis(...)> );
```

For example, the following JSL script adds test details for an effect to the report window:

```
dt = Open( "$SAMPLE_DATA/Dogs.jmp" );
manObj = dt << Fit Model(
    Y( logHist0, logHist1, logHist3, logHist5 ),
    Effects( dep1, drug, drug * dep1 ),
    Personality( "MANOVA" ),
    Run Model
);
manObj << Response Function( "Contrast" );
manObj << (Response["Contrast"] << (Effect["Whole Model"] <<

/* send the Test Details message to the Whole Model outline
under Contrast in the report window */
Test Details( 1 )));
```

```
/* send the Test Details message to response 1 (Contrast)
and effect 3 (drug*dep1) */
manObj << (Response[1] << (Effect[3] << Test Details( 1 )));
```

Send Function

Tip: Recall that the << operator is equivalent to the Send() function.

When you have multiple responses, you can send messages to a specific response column's fit.
Use the following JSL (where *responseName* is the specific response):

```
fitObj << ( responseName << {options, ...});
```

The second Send() function finds the named response and sends the list of messages to it.

Note: If you send the messages directly to the *fitObj* with a single Send() function, the
messages are sent to all responses.

In the following script, the last line sends a message to the AICc report for the ABRASION
response:

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
fitObj = dt << Fit Model(
  Y( :ABRASION, :MODULUS, :ELONG, :HARDNESS ),
  Effects(
    :SILICA & RS,
    :SILANE & RS,
    :SULFUR & RS,
    :SILICA * :SILICA,
    :SILANE * :SILICA,
    :SILANE * :SILANE,
    :SULFUR * :SILICA,
    :SULFUR * :SILANE,
    :SULFUR * :SULFUR
  ),
  Personality( "Standard Least Squares" ),
  Run
);
/* shows up under the Studentized Residuals plot*/
fitObj << (:ABRASION << {AICc( 1 )});
```

To send messages to an individual effect, nest even further:

```
fitObj << ( responseName << ((effectName) << effectOption ));
```

Standard Least Squares

For Standard Least Squares models that contain only fixed effects and more than one Y response, you can choose to fit the models for the Y responses together or separately. If some of the rows have missing values, the following apply:

- In a script, the Y responses are fit together by default. The model fits each Y using only those rows that are nonmissing for all of the Y variables. For example, a row that includes one or more missing Y values is *excluded* from the model. You can explicitly include Run("Fit Together") in a script to get the same result.

- If you include the Run("Fit Separately") option, the model fits each Y using all rows that are nonmissing for that particular Y. For example, a row that includes one or more missing Y values is *included* in the model.

The following script fits the Y responses together:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Fit Model(
    Y( :height, :weight ),
    Effects( :sex ),
    Personality( "Standard Least Squares" ),
    Emphasis( "Minimal Report" ),
    Run // or Run( "Fit Together" )
);
```

The following script fits the Y responses separately:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Fit Model(
    Y( :height, :weight ),
    Effects( :sex ),
    Personality( "Standard Least Squares" ),
    Emphasis( "Minimal Report" ),
    Run( "Fit Separately" )
);
```

When running the model in JMP, the user is prompted to select fitting the responses separately or together. The Fit Model launch window contains a Fit Separately option, which is deselected by default. For more information about how missing Y values are handled, see *Fitting Linear Models*.

Note: In models that contain a random effect, Y values are fit separately by default.

Formula Depot

Sending commands via JSL to a Formula Depot platform report requires that you first create an empty Formula Depot. Once you have a reference to the Formula Depot report, you can send commands to it. The following script opens a formula depot report and adds a model to the depot.

```
dt = Open( "$SAMPLE_DATA/Liver Cancer.jmp" );
fd = Formula Depot(); // open a new Formula Depot
obj1 = dt << Run Script( "Lasso Poisson, Validation Column" );
model = obj1 << (Fit[1] << Publish Prediction Formula );

/* Publish a model to the Formula Depot. The
following script opens a formula depot report, adds a model to
the depot, generates scoring code for the formula, saves the code to a file,
and closes all windows.*/
case_name = "fitLS_prediction";
dest_dir = "c:\"; //CHANGE ME to a better location
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Print( "Running " || case_name );
fm_ls = Fit Model(
    Y( :weight ),
    Effects( :age, :sex, :height ),
    Personality( "Standard Least Squares" ),
    Emphasis( "Minimal Report" ),
    Run
);
fd = fm_ls << Publish Prediction Formula;
Try(
    Print( "Generate Python Code" );
    code = fd << Generate Python Code( No Editor );
    Save Text File( dest_dir || case_name || ".py", code );
    Print( "Done." );
,
    Print( "Code generation failed for " || case_name || ", reason: " );
    Show( exception_msg );
);
fdw = Window( "Report: Formula Depot" );
fdw << Close Window( No Save );
Close( dt, No Save );
```

Graph Builder

The following is an example of the basic structure of a Graph Builder script:

```
Graph Builder(
```

```
     Size( x, y ),
     // Use any commands from the red triangle menu - Size is an example
     Variables( role( column ), ...), //Required
     Elements( element name( element options, ...), //Required
     Data Filter( ... ),
     // Use any display customizations here - Data Filter is an example
     );
```

The following is an example of a simple Graph Builder script that creates a bar chart:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Graph Builder(Variables(
  X( :age ),
  Y( :weight ) ),
Elements( Bar( X, Y, Legend( 8 ) ) )
// Legend is not required, but provides an internal legend ID that you can use
  for later customization
);
```

The following is a more detailed Graph Builder script that creates a bar chart:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Graph Builder(
Size( 373, 332 ),
Show Control Panel( 0 ),
Variables(
  X( :age ),
  Y( :weight ),
  Y( Transform Column( "weight25",
        Formula( Col Quantile( :weight, 0.25, :age ) )
        ),
     Position( 1 )
/* By default each Y variable gets its own axis, but the Position(1)
option keeps each additional Y in the first slot and merges the axes. */
  ),
  Y( Transform Column( "weight75", Formula( Col Quantile( :weight, 0.75, :age
  ) ) ),
     Position( 1 )
  )
),
Elements(
  Bar( X, Y( 2 ), Y( 3 ), Legend( 10 ), Bar Style( "Range" ) ),
/* This bar element uses only the second and third Y variables and
draws a Range bar between the values.*/
  Bar( X, Y( 1 ), Legend( 8 ),
     Bar Style( "Float" ),Summary Statistic( "Median" )
// This bar element uses the first Y variable and draws its median with the
  float style, which is a short horizontal line
```

```
      ),
      Points( X, Y( 1 ), Legend( 9 ) )
   // The Points element shows the raw data as markers.
      ),
      SendToReport(
   // Contains display customizations
         Dispatch( {}, "400", ScaleBox,
   // 400 is the fixed name
            {Legend Model( 10,
            Level Name( 0, "1st to 3rd quartiles",
            Item ID( "Mean(weight25)..Mean(weight75)", 1 )
   // Renames and changes the color for legend 10 (the Range bar)
         ),
            Properties( 0, {Fill Color( 32 )},
            Item ID( "Mean(weight25)..Mean(weight75)", 1 )
   // Changes the color for legend 10 (the range bar)
         )
            ), Legend Model( 9,
            Properties( 0, {Marker Size( 4 )}, Item ID( "weight",1 ) )
   // Changes the marker size for legend 9 (the Points element)
         )}
      ),
      Dispatch( {}, "graph title", TextEditBox,
         {Set Text( "Weights with quartile ranges" )}
   // Changes the graph title
      ),
      Dispatch( {}, "Y title", TextEditBox, {Set Text( "weight" )} ),
   // Changes the Y axis title
      Dispatch( {}, "400", LegendBox,
         {Legend Position( {10, [0], 8, [1], 9, [-1]} )}
   // Hides the legend item for legend 9 by giving it a position of -1
         )
      )
      );
```

Neural and Neural Net

The Neural platform replaces the Neural Net platform. Scripts that use Neural Net are
deprecated and might not work in the future. It is recommended to use the Neural platform
for new scripts and update any old scripts using Neural Net to Neural.

Partial Least Squares and PLS

The Partial Least Squares (PLS) platform has replaced the PLS platform. Scripts that use PLS might not work in the future. It is recommended to use the Partial Least Squares platform for new scripts and update any old scripts using PLS to Partial Least Squares.

Process Capability

The Process Capability platform has replaced the Capability platform. Scripts that use Capability might not work in the future. It is recommended to use the Process Capability platform for new scripts and update any old scripts using Capability to Process Capability.

Specification Limits in JSL Scripts

Specification limits can be read from JSL scripts, column properties, or from a specification limits data table. As an example of reading in specification limits from JSL, consider the following example:

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Process Capability(
  Process Variables( :OZONE, :CO, :SO2, :NO ),
  Spec Limits(
    OZONE( LSL( 0.075 ), Target( 0.15 ), USL( 0.25 ) ),
    CO( LSL( 5 ), Target( 7 ), USL( 12 ) ),
    SO2( LSL( 0.01 ), Target( 0.04 ), USL( 0.09 ) ),
    NO( LSL( 0.01 ), Target( 0.025 ), USL( 0.04 ) )
  )
);
```

Using a Limits Data Table and JSL

There is no extra syntax needed to differentiate between the two table types (wide and tall) when they are read using JSL. This example is based on the CitySpecLimits.jmp sample data table. It places the specification limits data table inside an Import Spec Limits() expression.

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Process Capability(
  Process Variables( :OZONE, :CO, :SO2, :NO ),
  Spec Limits(
    Import Spec Limits(
      "$SAMPLE_DATA/CitySpecLimits.jmp"
)));
```

Scatterplot 3D

If your machine supports acceleration, and you want to display the 3-D scatterplot faster, you can turn on the Use Hardware Acceleration option, or in JSL, specify the Scene3DHardwareAcceleration global variable. A value of 1 turns it on, and a value of zero turns it off.

```
dt = Open( "$SAMPLE_DATA/Solubility.jmp" );
Scene3DHardwareAcceleration = 1;
dt << Scatterplot 3D(
   Y( :"1-Octanol"n, :Ether, :Chloroform, :Benzene, :Carbon Tetrachloride,
   :Hexane )
);
```

Text Explorer

By default, text is converted to all lowercase in the Text Explorer platform. You can use the "From Source" keyword in the Save Regex Column message to save the results of a custom regex that preserve the original case of the text. The first argument is the name of the column that is created in the data table. The second argument is the string "From Source" or "From Result". The default behavior is equivalent to specifying the "From Result" keyword.

```
dt = Open( "$SAMPLE_DATA/Aircraft Incidents.jmp" );
obj = dt << Text Explorer(
   Text Columns( :Airport Name),
   Tokenizing( "Regex" ),
   Save Regex Column( "Output Column", "From Source" )
);
obj << Close Window();
```

Explore Outliers

The Explore Outliers platform provides options to identify, explore, and manage outliers in your univariate or multivariate data.

In the Probe.jmp sample data table, several columns contain outlier values of 9999. Many industries use nines as a missing value code. The following example selects VDP_PCOLL, VDP_PINNBASE, and VDP_PINPBASE, the columns with the highest number of 9s, and adds the Missing Value Codes column property to the columns. The script then rescans the columns and removes these columns from the report.

```
dt = Open( "$SAMPLE_DATA/Probe.jmp" );
obj = Explore Outliers(
   Y(
      :VDP_M1,
```

```
        :VDP_M2,
        :VDP_NBASE,
        :VDP_NEMIT,
        :VDP_NENBNI,
        :VDP_NSINK,
        :VDP_PBASE,
        :VDP_PBL,
        :VDP_PCOLL,
        :VDP_PEMIT,
        :VDP_PINNBASE,
        :VDP_PINPBASE,
        :VDP_PSINK,
        :VDP_SICR
    ),
    Quantile Range Outliers( 1 ),
    Show only columns with outliers( 1 ),
);
obj << Add Highest Nines to Missing Value Codes( :VDP_PCOLL, :VDP_PINNBASE,
    :VDP_PINPBASE );
obj << Rescan;
```

Scripting-Only Messages and Arguments by Platform

Some messages and arguments that can be sent to a platform object are scripting-only. This means that these options do not have corresponding options in JMP launch windows or reports. However, note that scripting-only options can appear as Preferences.

This section describes scripting-only messages and arguments for specific platforms.

Tip: For more information about any of these messages or arguments, and for specific examples, go to the Scripting Index (**Help** > **Scripting Index**).

- "Bivariate Script-Only Message"
- "Choice Script-Only Message"
- "Clustering Script-Only Messages"
- "Control Chart Script-Only Message"
- "Control Chart Builder Script-Only Messages"
- "Cumulative Damage Script-Only Message"
- "Custom Profiler Script-Only Messages"
- "Distribution Script-Only Messages"
- "Fit Life by X Script-Only Messages"
- "Fit Model Script-Only Messages"
- "Fit Parametric Survival Script-Only Messages"
- "K Nearest Neighbors Script-Only Messages"
- "Latent Class Analysis Script-Only Message"
- "Life Distribution Script-Only Messages"
- "Multivariate Script-Only Messages"
- "Naive Bayes Script-Only Messages"
- "Neural Script-Only Messages"
- "Pareto Plot Script-Only Message"
- "Partition Script-Only Messages"
- "Process Screening Script-Only Messages"
- "Surface Plot Script-Only Messages"
- "Text Explorer Script-Only Messages"
- "Time Series Script-Only Messages"

Bivariate Script-Only Message

The Fit Where message for Bivariate objects fits a curve or line to a single level of a categorical variable.

```
biv_object << Fit Where( WHERE_clause );
```

Choice Script-Only Message

When estimating the parameters in the Choice platform, you can set the acceptable criterion for convergence using this JSL function:

```
Convergence Criterion( fraction );
```

Clustering Script-Only Messages

The following messages for a hierarchical cluster object are available only in JSL:

```
// returns the column names in cluster order (for two-way clustering)
hier_cluster_object << Get Column Names

// returns a vector of cluster numbers
hier_cluster_object << Get Clusters

/* returns a vector of the display position for each row in the cluster,
with missing values for undisplayed rows */
hier_cluster_object << GetDisplayOrder

/* returns a vector of the display position for each column
(for two-way clustering)*/
hier_cluster_object << GetColumnDisplayOrder

// returns the distance matrix used for hierarchical clustering
hier_cluster_object << Get Distance Matrix
```

KMeans Clustering

The following message for a KMeans cluster object is available only in JSL. The message returns the mean and standard deviation for each variable within each cluster.

```
kmeans_cluster_object << Get Statistics
```

Control Chart Script-Only Message

The following argument for the Control Chart() function is available only in JSL. The message includes points that are excluded in the moving range calculations.

```
Use Excluded Points on MR( Boolean )
```

Control Chart Builder Script-Only Messages

The following arguments for the `Control Chart Builder()` function are available only in JSL.

```
// reruns all tests and alarms
Rerun All Tests

// sets the width and height of the control chart
Size( width, height )

// specifies the sigma multiplier
KSigma()
```

Cumulative Damage Script-Only Message

The following message for a Cumulative Damage object is available only in JSL. It returns a list of model fit results.

```
cumulative_damage_object << Get Results
```

Custom Profiler Script-Only Messages

The following messages for Custom Profiler objects are available only in JSL:

```
// returns the current value of the objective function
profiler_object << Get Objective

// set the formula for the objective function
profiler_object << Objective Formula

// returns the formula for the objective function as an expression
profiler_object << Get Objective Formula
```

Distribution Script-Only Messages

The following message for a Distribution object is available only in JSL. The message creates an array of handles to the fitted distributions, so you can send commands to specific distributions:

```
dist_object << ( Fit Handle[n] << ...);
```

You can also get a reference to a fitted distribution directly if you send the Fit Distribution message to a Distribution object. This enables you to send commands to specific distributions:

```
fit_object = dist_object << Fit Distribution( distribution_name );
```

```
fit_object << ...;
```

The following message for a Distribution object is available only in JSL. The message specifies that PpK labeling be used in the Process Capability reports within the Distribution report object.

```
dist_object << PpK Capability Labeling( 1 );
```

Fit Life by X Script-Only Messages

The following messages for a Fit Life by X object are available only in JSL:

```
/* set scriptable options within the profilers in different sections
of the output */
fit_life_object << Set Scriptables( ... );

/* return the estimates, standard errors, covariance matrix,
and convergence results for each distribution fit */
fit_life_object << Get Results
```

Fit Model Script-Only Messages

The following section describes scripting-only messages for MANOVA, Generalized Linear Models, Nominal and Ordinal Logistic, and Standard Least Squares.

MANOVA

The following message for a Fit MANOVA object is available only in JSL:

```
fit_model_object << ( Response[1] << (Effect[1] << ... ) );
```

Where the third message could be one of the following:

```
Test Details // show or hide the test details for an individual effect
Centroid Plot // show or hide the centroid plot for an individual effect
Save Canonical Scores // save the canonical scores for an individual effect
Contrast /* run a customized F test contrasting different levels for
        an effect in the model*/
```

The following example saves canonical scores for the Response Function, which in this data, is Sum:

```
dt = Open( "$SAMPLE_DATA/Dogs.jmp" );
obj = dt << Fit Model(
  Y( :LogHist0, :LogHist1, :LogHist3, :LogHist5 ),
  Effects( :drug, :dep1, :drug * :dep1 ),
  Personality( "Manova" ),
  Run( Response Function( Sum ) )
```

```
  );
  obj << (Response[1] << (Effect[1] << Save Canonical Scores) );
```

Generalized Linear Models

The following message for a Fit GLM object is available only in JSL. The message saves the
parametric formula to a new column in the data table

```
  Parametric Formula( )
```

Nominal and Ordinal Logistic

The following messages for Fit Nominal Logistic and Fit Ordinal Logistic objects are available
only in JSL:

```
  // create a SAS DATA step to score the data
  fit_model_object << Get SAS Data Step

  // create SAS code that you can register in the SAS Model Manager
  fit_model_object << Get MM SAS Data Step
```

Standard Least Squares

The following JSL messages return the requested item from the fitted model, such as variance
components, *p*-values, parameter estimates, and so on:

```
  fit_model_object << Get Variance Components( );
  fit_model_object << Get Effect Names( );
  fit_model_object << Get Effect PValues( );
  fit_model_object << Get Estimates( );
  fit_model_object << Get Parameter Names( );
  fit_model_object << Get Random Effect Names( );
  fit_model_object << Get Std Errors( );
  fit_model_object << Get X Matrix( );
  fit_model_object << Get XPX Inverse( );
  fit_model_object << Get Y Matrix( );
```

The following message for Standard Least Squares objects is available only in JSL:

```
  obj << Get SQL prediction expression;
```

The following example saves prediction formulas as SQL expressions and outputs them to the
log:

```
  dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
  obj = dt << Fit Model(
    Y( :ABRASION, :HARDNESS ),
    Effects( :SILICA, :SILANE, :SULFUR ),
    Personality( "Standard Least Squares" ),
    Run
```

```
);
code = obj << Get SQL Prediction Expression;
```

Fit Parametric Survival Script-Only Messages

The following JSL messages return the requested item from the fitted model, such as
parameter names, parameter estimates, standard error, and so on:

```
survival_object << Get Parameter Names
survival_object << Get Estimates
survival_object << Get Std Errors
survival_object << Get Effect Names
survival_object << Get Effect PValues
```

K Nearest Neighbors Script-Only Messages

The following JSL message constructs a script to create a prediction formula column and
returns it to the log.

```
knn_object << (Response[number] << Get Prediction Formula)
```

The following JSL message returns the value of the best K neighbors.

```
knn_object << Response[number] << Get Best K
```

Latent Class Analysis Script-Only Message

The following JSL message constructs a script to create probability formula columns and
returns them to the log.

```
latent_class_object << (Fit[number] << Get Probability Formulas)
```

The following example sets a random seed in order for it to be reproducible.

```
dt = Open( "$SAMPLE_DATA/Car Poll.jmp" );
obj = dt << Latent Class Analysis(
  Y( :sex, :marital status, :country, :size, :type ),
  Set Random Seed( 12345 ),
  Number of Clusters( 3 ),
);
```

Life Distribution Script-Only Messages

The following JSL messages prevent a specified plot from appearing in the report, or return the requested item from the fitted model, such as results, estimates, or formulas.

```
life_dist_object << Suppress Plot(plot_name)
life_dist_object << Get Results
life_dist_object << Get Estimates
life_dist_object << Get Formula
```

Multivariate Script-Only Messages

The following message for a Multivariate object is available only in JSL:

```
// create SAS PROC Mixed code to run similar estimation methods using SAS
multivariate_object << Create SAS Job( );
multivariate_object << Get Correlation Matrix;
multivariate_object << Get Inv Correlation Matrix;
```

Naive Bayes Script-Only Messages

The following messages for a Naive Bayes object are available only in JSL:

```
naive_bayes_object << Get Probability Formulas;
naive_bayes_object << Prior Bias( <fraction=0.5> );
```

Neural Script-Only Messages

Note: The Neural platform replaces the Neural Net platform. Scripts that use Neural Net are deprecated and might not work in the future. It is recommended to use the Neural platform for new scripts and update any old scripts using Neural Net to Neural.

The following message for Neural objects is available only in JSL:

```
// return the number of models used for boosting
neural_object << Get NBoost
```

The following JSL messages for Neural Fit objects return the requested item from the fitted model, such as entropy RSquare, generalized RSquare, misclassification rates, and so on:

```
neural_object << ( Fit[n] << Get RSquare Training )
neural_object << ( Fit[n] << Get RSquare Validation )
neural_object << ( Fit[n] << Get RSquare Test )
neural_object << ( Fit[n] << Get Gen RSquare Training )
```

```
neural_object << ( Fit[n] << Get Gen RSquare Validation )
neural_object << ( Fit[n] << Get Gen RSquare Test )
neural_object << ( Fit[n] << Get Misclassification Rate Training )
neural_object << ( Fit[n] << Get Misclassification Rate Validation )
neural_object << ( Fit[n] << Get Misclassification Rate Test )
neural_object << ( Fit[n] << Get Average Log Error Training )
neural_object << ( Fit[n] << Get Average Log Error Validation )
neural_object << ( Fit[n] << Get Average Log Error Test )
neural_object << ( Fit[n] << Get RMS Error Training )
neural_object << ( Fit[n] << Get RMS Error Validation )
neural_object << ( Fit[n] << Get RMS Error Test )
neural_object << ( Fit[n] << Get Average Absolute Error Training )
neural_object << ( Fit[n] << Get Average Absolute Error Validation )
neural_object << ( Fit[n] << Get Average Absolute Error Test )
neural_object << ( Fit[n] << Get ROC Area Training )
neural_object << ( Fit[n] << Get ROC Area Validation)
neural_object << ( Fit[n] << Get ROC Area Test)
neural_object << ( Fit[n] << Get Confusion Matrix Training)
neural_object << ( Fit[n] << Get Confusion Matrix Validation)
neural_object << ( Fit[n] << Get Confusion Matrix Test)
neural_object << ( Fit[n] << Get Confusion Rates Training)
neural_object << ( Fit[n] << Get Confusion Rates Validation)
neural_object << ( Fit[n] << Get Confusion Rates Test)
neural_object << ( Fit[n] << Get Seconds)
```

Pareto Plot Script-Only Message

The following message for a Pareto Plot object is available only in JSL:

```
No Plot( Boolean );
// close the outline for the Pareto plot
```

Partition Script-Only Messages

The following JSL messages for Partition objects return the requested item from the fitted
model, such as entropy RSquare, generalized RSquare, misclassification rates:

```
partition_object << Get RSquare Training
partition_object << Get RSquare Validation
partition_object << Get RSquare Test
partition_object << Get Gen RSquare Training
partition_object << Get Gen RSquare Validation
partition_object << Get Gen RSquare Test
partition_object << Get Misclassification Rate Training
partition_object << Get Misclassification Rate Validation
partition_object << Get Misclassification Rate Test
```

```
partition_object << Get ROC Area Training
partition_object << Get ROC Area Validation
partition_object << Get ROC Area Test
partition_object << Get Average Log Error Training
partition_object << Get Average Log Error Validation
partition_object << Get Average Log Error Test
partition_object << Get RMS Error Training
partition_object << Get RMS Error Validation
partition_object << Get RMS Error Test
partition_object << Get Average Absolute Error Training
partition_object << Get Average Absolute Error Validation
partition_object << Get Average Absolute Error Test
partition_object << Get Seconds
partition_object << Get SAS Data Step
partition_object << Get Tolerant SAS Data Step
partition_object << Get MM SAS Data Step
partition_object << Get MM Tolerant SAS Data Step
```

Process Screening Script-Only Messages

The following JSL message for Process Screening saves the row states for the table of
processes.

```
process_screen_object << RowStates
```

Surface Plot Script-Only Messages

The following messages for a Surface Plot object can also be controlled in JMP, but might not
have corresponding options in the red triangle menu of another platform such as Profiler:

```
/* Isosurface uses three independent variables as the X, Y, and Z axes and
   simultaneously plots the surfaces for the dependent variables on the plot.
   Sheet has one Z response that is dependent upon two independent X, Y
   variables.*/
surface_plot_object << Mode( "Isosurface" | "Sheet" )

// set the Z axis by referencing the column
surface_plot_object << Set Z Axis( col, Current Value( n ) )

// set the independent variable axes by referencing each individual column
surface_plot_object << Set Variable Axis( col, Current Value( n ) )

// identify up to four response columns for plotting overlaid points
surface_plot_object << Response( ... )
```

```
// assign formulas to columns in the sheets in a specified order
surface_plot_object << Formula( ... )
```

Text Explorer Script-Only Messages

The following JSL messages for a Text Explorer object are available only in JSL:

```
/* specify the minimum number of occurrences a phrase must have
to be included in the phrase list */
text_explorer_object << Minimum Frequency for Phrase( n )

// alias for Save Document Term Matrix()
text_explorer_object << Save Indicators for Most Frequent Words( ... )

// alias for Save Term Table()
text_explorer_object << Save Word Table( ... )

// alias for Score Terms by Column()
text_explorer_object << Score Words by Column( ... )

// replace default regular expressions used in Regex tokenizing
text_explorer_object << Set Regex( ... )

// create a new column with tokenized results
// "From Source" keyword uses the capitalization of the source strings
text_explorer_object << Save Regex Column( string, <"From Source"> )

// add user-defined delimiter characters, specified in a single string
text_explorer_object << Add Delimiters( string )

// replace default delimiter characters, specified in a single string
text_explorer_object << Set Delimiters( string )
text_explorer_object << Add Stop Words( list )
text_explorer_object << Add Phrases( list )
text_explorer_object << Add Stem Exceptions( list )

// add a list of words that are always allowed to be stemmed
text_explorer_object << Add Stem Overrides( list )

// add a list of words to remove from the list of stop words
text_explorer_object << Add Stop Word Exceptions( list )

// add a list of phrases to remove from the list of specified phrases
text_explorer_object << Add Phrase Exceptions( list )

// add a list of pairs of words to recode
```

```
text_explorer_object << Add Recodes( list )

// add a list of recoded text strings to remove
text_explorer_object << Add Recode Exceptions( list )
text_explorer_object << Terms Alphabetical( Boolean )
text_explorer_object << Phrases Alphabetical( Boolean )

// set the width of the word cloud in pixels
text_explorer_object << Cloud Width( pixels )

// supply a function for custom stemming
text_explorer_object << Custom Stemmer( function( {string, dot}, ... ) )
```

Time Series Script-Only Messages

The following JSL messages for a Time Series object are available only in JSL:

```
// return a list of model results named by model description
time_series_object << Get Models

// return a list of model results named by model specification
time_series_object << Get Model Specs
```

Display Trees
Create and Interact with Windows

After you learn the basics of scripting, you're ready to manipulate JMP reports and create new JMP windows. This chapter focuses on the following topics:

- navigating JMP display trees to manipulate items in report windows
- building new windows with custom results
- building modal windows
- interacting with the script editor
- converting deprecated `Dialog()` scripts to `New Window()` scripts

Display Tree Compatibility Issues

JMP platforms are generally stable from release to release. However, sometimes the structure of reports changes in major releases. When this happens, old scripts can fail because of the new structure. To avoid the continual updating of old scripts, write scripts so that they avoid dependence on the report structure. Using a string to index a box, followed by a numbered subscript to pick a specific child box, is most reliable across releases. See "Create a Reference by Subscripting" on page 526. See "Compatibility Notes" on page 913 for compatibility issues in JMP 16.

Note: This chapter shows how to create and use the most common display boxes. All display boxes are documented in the JMP Scripting Index, which is available in the Help menu.

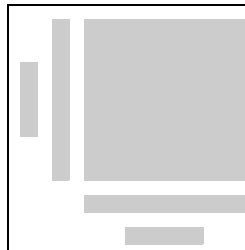# Contents

Navigate JMP Reports

As shown in Table 11.1, reports in JMP are built in a hierarchical manner by nesting and concatenating different types of rectangular boxes called *display boxes*. This section shows you how to navigate reports, extract data from them, and customize them.

Table 11.1 Common Display Boxes in JMP Reports

Outline Box() creates an outline hierarchy.

Picture Box() combines axes, frames, and labels to make graphs.

Table Box() is a special H List Box() whose elements are string and number columns.

string col box   number col boxes

The hierarchical (or parent-child) relationships between display boxes help determine the sizing. The outermost display box is the parent; the inner display boxes are children. When a report is displayed, the parent display box detects the size of the last child box (the leaf) and then detects the size of the child box that precedes the last child box. The size is detected for each box on the way back up the tree. Then the positioning of the boxes takes places from the top down so that the boxes can be shown in an orderly arrangement.

The hierarchical (or parent-child) relationships between display boxes form a tree of display boxes, called the *display tree*. For (parent) boxes that contain other (child) boxes, the parent box's size is determined by the sizes of its child boxes rather than some fixed size, which keeps the whole display resizable. Similarly, positions of boxes are determined by positions of the boxes before them rather than using fixed positions. In tree terminology, the sizing is bottom-up and the positioning is top-down.

Examples of Common Display Boxes

Figure 11.1 shows a Bivariate report that consists of several common display boxes.

Figure 11.1 Display Boxes in a Report

The Bivariate Fit outline box contains a picture box, and a text edit box.

The Linear Fit line is in a picture box. The red triangle is in a popup box.

The Linear Fit outline box contains a text edit box and outline boxes for the child reports as shown.

The Analysis of Variance outline box contains a table box, which contains a string column box and number column boxes that have titles.

The Parameter Estimates outline box contains a table box, which contains a string column box and number column boxes.

Run the following script to create the report:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :weight ), X( :height ), Fit Line() );
```

View the Display Tree

View the report structure to see how display boxes are arranged in the display tree.

1. Run the following script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
    Y( :weight ),
    X( :height ),
    Fit Mean( {Line Color( {57, 177, 67} )} )
);
```

2. Right-click the top disclosure icon and select **Edit > Show Tree Structure**.

Figure 11.2 Top Disclosure Icon

In the Show Tree Structure window, you see the display box structure of the report as nested outline boxes. Each outline box corresponds to a display box in the original report.

3. Click the Bivariate graph. In Figure 11.3, the frame box that contains the graph is highlighted.

Figure 11.3 Show Tree Structure
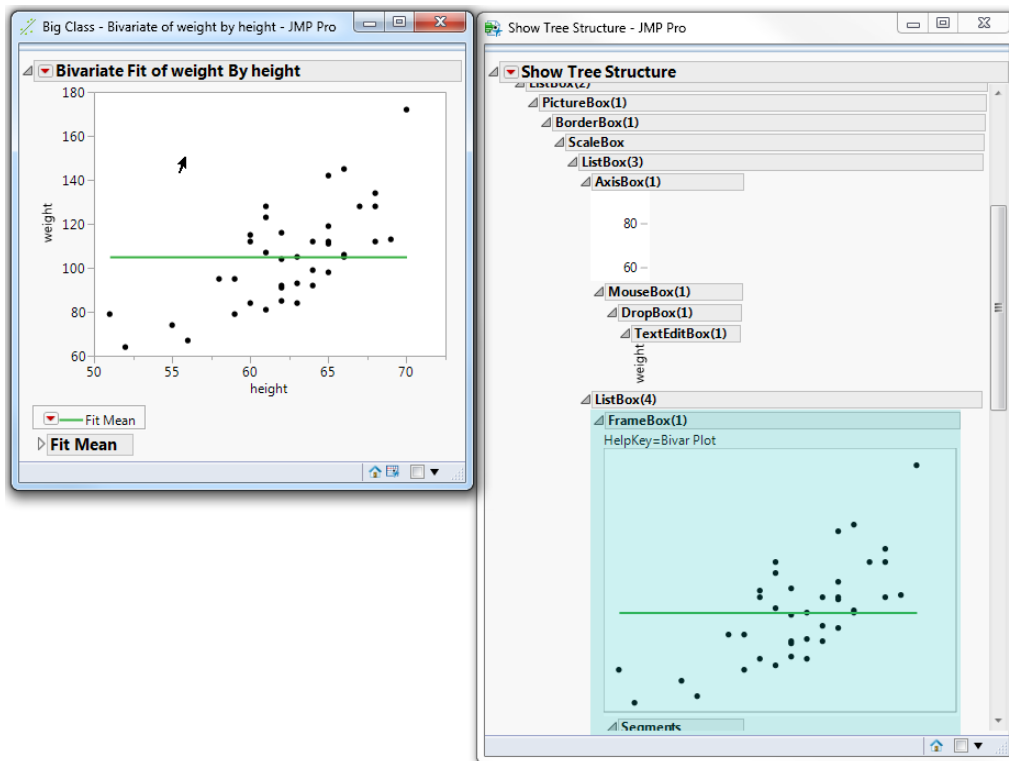

You can also obtain the tree structure through a script. Send the Show Tree Structure() message to any report. Or, send the message to a piece of the report (any display box object) to see the tree structure for just that part of the report.

The newer JMP tree structure provides a flat view of the display box tree rather than an outline view. To view the classic structure, press Shift before you right-click the disclosure icon and then select **Edit > Show Tree Structure**. Note the following differences between the newer and classic tree structure views:

•   The classic tree structure is not linked to the original report; you cannot highlight specific portions of the tree.

•   Siblings appear to the left or right of each other in the classic tree structure. In the newer tree structure, siblings are above or below each other.

Figure 11.4 Partial View of the Classic Tree Structure

Show Display Box Properties

There is an easy way to format reports before you save them to a script. You can edit the display box properties such as color and position in the Show Tree Structure window. In the report window, select **Edit > Properties** (Windows) or **View > Show Properties** (macOS). Click the node that you want to edit.

In Figure 11.5, the Summary of Fit statistics were selected below the graph. The Background Color and Text Color were then changed in the Properties. These changes are preserved when you save the report as a script or journal the report.

Figure 11.5 Example of Showing Properties for a Plot

Display Box Object References

When you launch a platform, the results are in two places: a visible report, made of display boxes, and an invisible object, called a platform, that accepts commands (or messages).

One of the messages that you can send a platform is the Report message. The Report message retrieves the display box that holds the report.

The red triangle menu at the top of the report is connected to the platform object and is the way you interactively send messages to the platform.

To fit the mean in the Bivariate platform, assign a reference to the Bivariate platform.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( // biv references the analysis layer
    X( height ),
    Y( weight ),
    Fit Mean,
    Fit Polynomial( 4 )
);
```

To get a reference to the display box that represents the view of the platform, use the `Report` message. For example, add this expression to the preceding script:

```
rbiv = biv << Report; // rbiv references the report layer
```

Now you can manipulate the display boxes in the report layer as described in the following sections.

See What You Can Do with a Display Box

The following example creates a Bivariate report and stores its reference in the `rbiv` report object. It then puts the `rbiv` report object in a frame box and assigns the `fbx` variable.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( // biv references the analysis layer
    Y( :weight ),
    X( :height ),
    Fit Mean( {Line Color( {57, 177, 67} )} )
);
rbiv = biv << Report; // assign rbiv to the report layer
fbx = rbiv[Frame Box( 1 )];
```
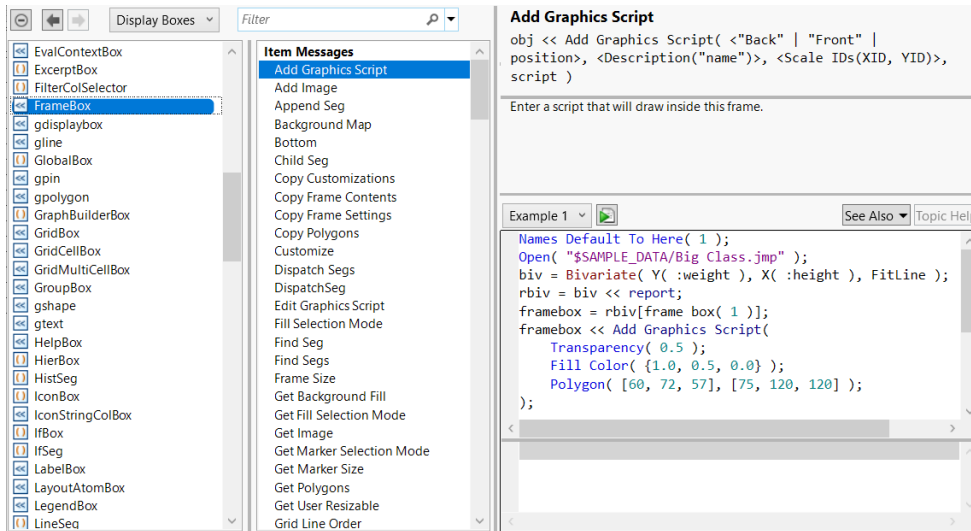
After fbx has been defined, you can send it a message.

```
fbx << Set Background Color( blue ); // colors the frame box in blue
```

The `Set Background Color` message works the same as right-clicking in the graph and selecting **Background Color > Blue**.

To find out which messages you can send to a frame box, select **Help > Scripting Index** and search for "frame box".

Figure 11.6 Scripting Index Display Boxes List

Running the Show Properties() function is an alternative to using the Scripting Index. The messages for the specified display box are printed to the log.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :weight ), X( :height ) );
Show Properties( biv );
    Show Points [Boolean] [Default On]
    Histogram Borders [Boolean]
    Fit Mean [Action](Fits a flat line at the mean.)
    ...
```

Many messages are the same as items in the pop-up menu for a given object. The next section, "Construct Custom Windows" on page 539, discusses messages for common display boxes in more detail.

Create a Reference by Subscripting

Use the subscript operator to navigate to a specific display box in the report. The best practice is to refer to the display box with an index number. The following expression identifies the second outline box in the display tree:

```
var = Outline Box[2];
```

Another way to navigate a report is to search for a text string. The following expression identifies the report that has the specified text string:

```
var = report[text];
```

This expression finds the display box in *rpt* that has the title *text*. Note that *text* must be the complete title, not just a substring of the title. The "*text*" argument can also be any expression that evaluates to a text string. This approach is less reliable than referring to the display box with a subscript because the string in a report can easily change.

Typically, you want to assign the identified part of the report to a variable so that you can send messages to it easily. For example, to find the Analysis of Variance outline box in the Bivariate report, use this expression:

```
var = rbiv["Analysis of Variance"];
```

The sections below describe other methods for using a subscript operator to locate display boxes.

Wildcard String

You can use a wildcard character (such as "?") with a substring to match the rest of the outline title. The wildcard character "?" represents places in the search string where you want to match any sequence of any characters. The following script searches for a string that ends with "Mean", finds "Fit Mean", and then opens the Fit Mean outline:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
  Y( :weight ),
  X( :height ),
  Fit Mean( {Line Color( {57, 177, 67} )} ),
  Fit Polynomial( 4 )
);
rbiv = biv << Report; // return a reference to the Bivariate report
out1 = rbiv["? Mean"]; // find "Fit Mean" in the Bivariate report
out1 << Close( 0 ); // open the Fit Mean outline
```

Box Type and Wildcard String

Another method for locating display boxes is to include the display box and wildcard search string in the substring operator.

```
var = rbiv[Display Box( "?text" )];
```

This expression finds the outline box that contains the specified text string in the rbiv report. The out1 variable assigns a reference to the outline box.

```
out1 = rbiv[Outline Box( "? Mean" )];
```

Box Type and Index Number

Another method for locating display boxes uses the *boxType* and an index number, *n*.

```
rpt[Display Box ( n )];
```

The expression above finds the *n*th of the specified display box of the type *boxType*. For example, this expression finds the first outline box in the `rbiv` report and assigns it to the `out1` variable:

```
out1 = rbiv[Outline Box( 1 )];
```

This method is less reliable across releases because the number of a display box might change.

Tip: To determine the index number of a display box, view the tree structure as described in "View the Display Tree" on page 521.

Nested Display Boxes

To reference a display box that appears several layers down in a report, use this format:

```
var = rpt[arg1][arg2][arg3][...];
```

The expression above matches the last argument to a display box that is *contained by* the next to last argument's outline node, which is in turn contained by the argument before the next to last, and so on. In other words, it is a way of digging down into the hierarchy of an outline tree to identify a nested display box.

If you want to identify an item nested several layers down in an outline box, use a subscript with more than one argument of any of the above types. After locating the first item, JMP looks *inside* that item for the next, and so on.

```
rpt[arg1][arg2][arg3] // finds the first item then the next starting after
    that location, and so on
rpt[arg1, arg2, arg3] // different syntax, same result
```

Note that you can string together subscripts that have text or index numbers. The expression below finds the "Parameter Estimate" inside the outline node "Polynomial Fit Degree=4" and selects the third display box in the first table of the outline node.

```
out = rbiv["Polynomial Fit Degree=4"]["Parameter Estimates"][1][3] << Select;
```

"Example of Creating a Report" on page 536 shows how to extract parameter estimates from the report.

Subscripting Display Boxes for Compatibility

JMP might add or remove display boxes in a report from release to release, causing numbered subscripts to choose a different box. Using a string to index a box, followed by a numbered subscript to pick a specific child box, is more reliable across releases.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
rp = dt << Bivariate(
    Y( :weight ),
    X( :height ),
```
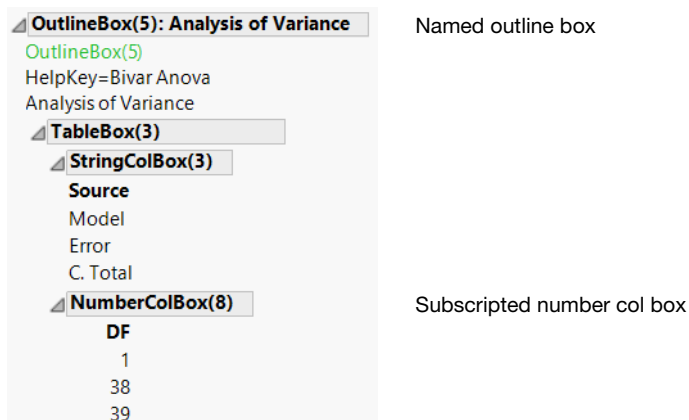
```
    Fit Line( {Line Color( {213, 72, 87} )} )
);
Print(
  "C.Total DF=",

  // string references the parent box
  Report( rp )[Outline Box( "Analysis of Variance" )][
  Number Col Box( 1 )][3] // subscript references the child box
);
```

Figure 11.7 Subscripted Display Box

The numbered subscripts (in the preceding example) are relative to the named box, which makes them less likely to change. However, this also means that the absolute numbers in Show Tree Structure can't be used directly.

Send Messages to Display Boxes

Use the display reference to send messages to the display elements using the Send or << operator. For example, if **out2** is a reference to an outline node, you can ask the node to close itself if it is open:

```
out2 << Close(); // close the outline node
```

Close() toggles an outline node back and forth between closed and open states, just like selecting items in the red triangle menu. You can include a Boolean argument (1 or 0) for "close or leave closed" or "open or leave open."

```
rbiv["Fit Mean"] << Close;    // toggle open and closed
rbiv["Fit Mean"] << Close( 1 ); // close or leave closed
rbiv["Fit Mean"] << Close( 0 ); // open or leave open
```

Tip: The Report Invalid Display Box Messages option in the General preferences helps you identify invalid display box messages.

- The preference is off by default. When a display box receives an invalid message, JMP ignores those errors and continues executing the script.

- When this preference is on, an error appears in the log.

This option can be useful during script development but can cause unwanted log messages for existing scripts.

The << Operator

You can send messages to display boxes using the send (<<) operator. The operator makes it clear when the script is evaluating child arguments and optional arguments. It also helps make it clearer which argument is an option and which is a script to run inside the graph.

Here is an example of sending a Graph Box a message:

```
win = New Window( "Messages",
    gb = Graph Box(
        Frame Size( 400, 400 ),
        X Scale( 0, 25 ),
        Y Scale( 0, 25 )
    )
);
gb << Background Color( "red" );
```

Nesting and Precedence

When you stack (or *nest*) messages, each message is evaluated left to right.

- The following expression sends message1 to box, then message2 to box, then message3 to box. Note that any of the messages can change box before the next message is sent.

  ```
  box << message1 << message2 << message3;
  ```

- The following expression sends message1 to box and gets the result. Message2 is sent to that result, and message3 is sent to the result of message2.

  ```
  ( (box << message1) << message2) << message3
  ```

- The result of message3 is assigned to the variable x.

  ```
  x = box << message1 << message2 << message3;
  ```

Here is an example of using nested messages:

```
win = New Window( "Messages", gb = Graph Box() );
```

```
// message 1 sets the background color
sz = gb << Background Color( "red" )

// message 2 saves the graph box as a PNG file
<<Save Picture( "$DOCUMENTS/red.png", "png" )

// message 3 sets the graph box background color to white
<<Background Color( "white" )

// message 4 returns the graph box size and prints it to the log
<<Get Size();
```

If you nest several messages, you might want to use parentheses to group the messages to be sure your script does what you want it to.

Customize Reports with the Send To Report and Dispatch Functions

The `Send To Report()` and `Dispatch()` functions provide a way for JMP to record display box modifications inside an analysis script in a self-contained way. For example, arguments within the two functions open and close outline nodes, resize graphics frames, or customize the colors in a graphics frame. You can also specify the arguments in separate JSL statements.

`Send To Report()` contains a list of options that change the display tree.

`Dispatch()` contains four arguments that change the display tree:

- The first argument is a list of outline nodes that need to be traversed to find the desired part of the display tree.

- The second and third arguments work together. The second argument is the name of a display element, and the third argument is the display element's type. These two arguments specify which particular part of the display tree is to be customized.

- The fourth argument is a list of options.

For example, open Big Class.jmp and run the attached Bivariate script. This generates a report with a fitted line. Open the Lack of Fit outline node, and close the Analysis of Variance outline node. Select **Save Script > To Script Window**. The following script appears in the script editor window:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
  Y( :weight ),
  X( :height ),
  Fit Line(),
  SendToReport(
    Dispatch(
      {"Linear Fit"},
      "Lack Of Fit",
```

```
        OutlineBox,
        {Close( 0 )} ),
      Dispatch(
        {"Linear Fit"},
        "Analysis of Variance",
        OutlineBox,
        {Close( 1 )}
      )
    )
  );
```

The `Send To Report()` function contains two `Dispatch()` functions that customize the default report.

- The first argument finds an outline node named "Linear Fit".
- The second and third arguments finds an outline box named "Lack of Fit" below the "Linear Fit" outline.
- The fourth argument is the argument to send to this outline box. In this case, the message is `Close(0)`, in other words, open the node.

Note: If there are several outline nodes with identical names, subscripts are assigned to them. For example, if you have a Bivariate analysis with two quadratic fits (resulting in identical titles), when you dispatch a command to the second fit, the subscript [2] is added to the duplicated title.

The best way to deal with `Send to Report()` and `Dispatch()` is to first create a customized report interactively. Save the report as a script and then examine the script that JMP generates. Remember: the best JSL writer is JMP itself.

Create a Journal

Creating a journal enables you to capture the contents of a JMP window at a fixed moment in time. Suppose that a JMP window contains a graph and selected check boxes. The journal shows the graph and the check boxes that were selected when you created the journal.

The following examples show what you can do with a journal.

Suppose that you start with a Bivariate report.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( weight ), X( height ) );
rbiv = biv << Report;
```

Journal the results:

```
rbiv << Journal Window;
```

Save the journal to a file:

```
rbiv << Save Journal(
  "$DOCUMENTS/test.jrn"
);
```

Note that the above example uses macOS pathname conventions. On Windows, you can use forward slashes (which are appropriate for all platforms) or backslashes (which are appropriate only for Windows).

When a script uses a By variable, the result is a list of references to the analysis results for each By group. In order to journal all By member parts of the report, you need to use parent messages to get to the top of the report.

For example, the following script creates a Bivariate report with a By group and then journals the entire report:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
  Y( weight ),
  X( height ),
  By( sex )
);
( ( Report( biv[1] ) << Parent ) << Parent ) <<
Save Journal( "test.jrn" );
```

Journals with Grid Lines

When axes are collapsed, grid lines disappear when the report is journaled, pasted, or saved as a PDF file. Saving the report from JMP as a picture file or a Microsoft PowerPoint presentation retains the grid lines in the graph.

Using Common Messages

Update a Numeric Column

To update a numeric column in a display box table, use the Set Values message.

```
win = New Window( "Numeric Column",
  num = Number Col Box( "Values", [9, 10, 11] )
);
num << Set Values( [1, 2, 3] );
```

The matrix argument specifies the new numbers for the table.

Add the Thousands Separator in a Numeric Column

To add the thousands separator in a numeric column, use the Set Format message.

```
New Window( "Example",
    ncb = Number Col Box( "Random Numbers",
        {Random Uniform(), Random Uniform() * 10,
        Random Uniform() * 100, Random Uniform() * 1000,
        Random Uniform() * 10000}
    )
);
ncb << Set Format(10,3, "Fixed Dec", "Use thousands separator");
```

Update a String Column

To update a string column in a display box table, use the Set Values message.

```
win = New Window( "String Column",
    str = String Col Box( "Values", {"a", "b", "c"} )
);
str << Set Values( {"A", "B", "C"} );
```

The list argument specifies the new strings for the table.

Wrap Text

Generally, JMP automatically wraps text within Text Box(). However, the default wrap point
can be overridden with a Set Wrap (*n*) message. The following script wraps the text at 200
pixels.

```
win = New Window( "Set Wrap",
    tb = Text Box(
        "Hover over a data point for more information."
    )
);
tb << Set Wrap( 200 );
```

Include Bullet Points in Text Boxes

You can add bullet points using the Bullet Point(1) message.

```
win = New Window( "Bullet List",
    text1 = Text Box( "Hover over a data point for more information." ),
    text2 = Text Box( "Draw a circle around a statistic for more information." )
);
text1 << Bullet Point( 1 );
text2 << Bullet Point( 1 );
```

Sending the Bullet Point(1) message to a text box places a bullet in front of the text and
indents subsequent lines within that text box.

You can also send the Bullet Point(1) message inline as shown in the following example:

```
text1 = Text Box( "Hover over a data point for more information." ), << Bullet
    Point( 1 ) ),
```

Blink a Selected Display Box

You can use `Select`, `Reshow`, and `Deselect` to blink the selection highlight on a display box.
As previously described, you can string together several << clauses to send an object several
messages in a row. The results read left to right; for example, here `Select` is done first, then
`Reshow`, then `Deselect`, then the other `Reshow`.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
    Y( :weight ),
    X( :height ),
    Fit Mean( )
);

// assign the bivariate analysis layer to rbiv
rbiv = biv << Report;

/* find outline box with the specified string in the
rbiv report and assigns it to the out2 variable */
out2 = rbiv[Outline Box( "? Mean" )];
out2 << Close( 0 ); // open the outline box

// find the first string col box in rbiv and
assign the scbx variable */
scbx = rbiv[String Col Box( 1 )];

Wait( .25 ); // wait .25 seconds
For( i = 0, i < 20, i++, // set i to 0, iterate through the loop 20 times

    /* select the string col box and reshow it,
    deselect it and reshow it */
    scbx << Select << Reshow << Deselect << Reshow
);
```

Add Shading or Borders to Table Headings

Headings are shaded and have column borders by default. To turn them off, use the `Set
Shade Headings (0)` message and the `Set Heading Column Borders (0)` message.
The following example shows the effect of turning shading and borders off:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
    meanHt = Mean( Height ),
    minHt = Min( Height ),
```

```
      maxHt = Max( Height )
   );
   win = New Window( "Summary Results",
      tb = Table Box(
         Number Col Box( "Mean", meanHt ),
         Number Col Box( "Min", minHt ),
         Number Col Box( "Max", maxHt )
      )
   );
   Wait( 2 );
   tb << Set Shade Headings( 0 );
   Wait( 1 );
   tb << Set Heading Column Borders( 0 );
   Wait( 2 );
   tb << Set Shade Headings( 1 );
   Wait( 1 );
```

Example of Creating a Report

Here is a script to build a report from start to finish.

1. Open a data table.

   ```
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
   ```

2. Launch a Bivariate platform and assign it to the platform reference biv.

   ```
   /* a reference to the platform */
   biv = dt << Bivariate( Y( weight ), X( height ) );
   ```

3. To find out what you can do with the Bivariate platform, search for Bivariate in the Objects list of the Help > Scripting Index. Here are a few options:

   ```
   biv << Fit Spline( 1000000 ) << Fit Mean;

   // degree 4, orange curve
   biv << Fit Polynomial( 4, {Line Color( RGBColor( 1, 0.5, 0 ) )});
   biv << ( Curve[1] << Line Color( red ) ); // first curve is red
   ```

4. Assign rbiv to the report layer:

   ```
   rbiv = biv << Report;
   ```

5. Get the frame box to a comfortable size and scroll up to the top to see the graph.

   ```
   rbiv[Frame Box( 1 )] << Size Window( 500, 700 );
   biv << Scroll Window( {0, 0} );
   ```

6. Get to work on the report. First, you need to create a reference, and then see what you can do with it.

   ```
   Show Properties( rbiv ); // show the properties of the report
   ```

Note: The log lists the messages to use with reports. Using `Show Properties()` is an alternative to the Scripting Index.

```
Close [Boolean]
GetOpen [Action] [Scripting Only]
SetOpen [Boolean] [Scripting Only]
...
```

---

7. Open the Fit Mean node of the outline:

```
rbiv["Fit Mean"] << Close( 0 );
```

`Close()` is a Boolean function (true or false), and passing 0 indicates that the outline is not closed.

8. Practice selecting some results (submit each line alone to see its result):

```
rbiv["Summary of Fit"] << Select;
rbiv["Parameter Estimates"] << Select;
rbiv["Analysis of Variance"] << Select;
rbiv["Polynomial Fit Degree=4", "Parameter ?", Column Box( "Estimate" )] <<
    Select; // dig down in the outline tree
rbiv << Deselect;
```

9. Change the format of the Estimate column in the 4-degree polynomial Parameter Estimates report.

```
pe = rbiv["Polynomial Fit Degree=4", "Parameter ?"];
ests = pe[Number Col Box( "Estimate" )];
ests << Set Format( 12, 6 );
```

The first argument to `Set Format` sets the column width by the number of characters to display. The second argument sets how many decimal places are shown in the table.

Figure 11.8 Applying Changes to a Report

before

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	-17.60008	84.82175	-0.21	0.8368
height	1.9521428	1.346006	1.45	0.1559
(height-62.55)^2	-0.220179	0.29996	-0.73	0.4678
(height-62.55)^3	0.0703948	0.035633	1.98	0.0561
(height-62.55)^4	0.0073899	0.004231	1.75	0.0895

after
Set Format(12,6)

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	-17.600085	84.82175	-0.21	0.8368
height	1.952143	1.346006	1.45	0.1559
(height-62.55)^2	-0.220179	0.29996	-0.73	0.4678
(height-62.55)^3	0.070395	0.035633	1.98	0.0561
(height-62.55)^4	0.007390	0.004231	1.75	0.0895

Note: You can even get a single number out of the table. For example, get the estimate for the cubic term by adding the following script to the preceding script:

```
terms = rbiv[Outline Box( "Parameter Estimates" )][String Col Box( "Term"
    )]; // find the Term column
ests = rbiv[Outline Box( "Parameter Estimates" )][Number Col Box(
    "Estimate" )]; // find the Estimate column
estimate = .; // in case there is no cubic term
Try(
    For( i = 1, i < 10, i++,

        // get(i) goes off the end of the table handled by Try
        If( Contains( terms << Get( i ), "^3" ),
         estimate = ests << Get( i );
            Break();
        )
    )
);
Show( estimate );
    0.070394822744608
```

10. Use a `For()` loop to count down to the row for the term that you want. The second argument to `For()` is a condition; as long as the condition tests true, looping continues. Here the test is "when the string in the Terms column is not "^3" and we have not reached the tenth row." As soon as the string *does* match, looping stops and i's value is the number for the matching row. You then use i as a subscript to the Get message on the Estimates column.

11. You can also get values from boxes as a matrix, which you can then use for further computations or write to a data table. The following example also shows how to make a data table:

```
// get the values in the Sum of Squares column as a matrix
myVector = rbiv[Table Box( 5 )][Number Col Box( "Sum of Squares" )] << Get as
    Matrix;

/* create a new column in Big Class.jmp named "Sum of Squares"
insert the values of myVector in the column */
dt << New Column( "Sum of Squares", Values( myVector ) );

// put the values of the fifth table box into a new data table
rbiv[Table Box( 5 )] << Make Data Table( "ANOVA table" );
```

12. Now adjust the scales on the axes.

```
rbiv[Axis Box( 1 )] << Min( 70 ) << Max( 170 ); // set the Y axis
rbiv[Axis Box( 2 )] << Min( 50 ) << Max( 70 );  // set X axis
```

13. Copy the graph at the top of the report. Note that you need to select the picture box that contains the graph; selecting just the graph would leave its axes behind.

```
rbiv[Picture Box( 1 )] << Copy Picture;
```

Construct Custom Windows

You can use constructor functions to create objects and construct your own displays. For example, you might prompt the user for information, create custom reports, create custom launch windows, or add check box and radio button items to a window.

Start with a New Window(), including a title in quotes, and then construct display boxes inside the window. Make a reference to the new window by assigning it a variable so that you can send messages to it. When display objects are created or referred to by JSL, they are freely shared references until they are copied into another display box or until you close the window and they disappear. When you plug a display object into another display tree, JMP makes a copy of it that the new box owns.

The names of all display box constructors end in "Box". There are similar objects that live inside graphics frames called display segs (segments such as markers and lines). The names of all display seg objects end in "Seg".

Example of Creating a Graph Box

The following example creates a new window that contains a graph box:

```
win = New Window( "Simple Earth Map",
  gb = Graph Box(
      X Scale( -180, -60 ),
      Y Scale( 20, 80 ), // end of graph box parameters
      <<Background Map( Images( "Simple Earth" ) ) // beginning of script
   )
);
```

The parameters set the x-axis and y-axis scale and the frame size. The Background Map message then shows the map. Note that the parameters for Graph Box(), X Scale and Y Scale, come first. After the first comma, the script begins. If you put the script in the middle of the parameters in a New Window() function, the script might run but not produce the results you expect. Scripts are also easier for the users to read if the parameters are at the top of the new window.

Extract Values from a Window

After the user selects options in a window, you can extract those selections from the window using the Return Result message, Get message, or Get Selected message.

Extract Values: Method One

To have a New Window() script automatically return the results after the user clicks OK, include the Return Result message after the Modal message.

```
win = New Window( "Set a Value",
    <<Modal,
    <<Return Result,
    Text Box( "Set this value" ),
    variablebox = Number Edit Box( 42 ),
    Button Box( "OK" ),
    Button Box( "Cancel" )
);
Write( win["variablebox"] ); // create a subscript to the variablebox variable
    33 // the user typed "33" in the number edit box
```

Extract Values: Method Two

Include the Get message to return 1 for a selected check box and 0 for a deselected check box. To view the selection, add a Show() expression at the end of the script.

```
win = New Window( "V List Box",
    <<Modal,
    V List Box(
        kb1 = Check Box( "a" ),
        kb2 = Check Box( "b" ),
        kb3 = Check Box( "c" )
    ),
    Button Box( "OK",
        val1 = kb1 << Get; // get the value of the first check box
        val2 = kb2 << Get;
        val3 = kb3 << Get;
    )
);
Show( val1, val2, val3 ); // return variables after window closes
    val1 = 1; // first and second checkboxes are selected
    val2 = 1;
    val3 = 0; // third checkbox is not selected
```

Extract Values: Method Three

Include the `Get Selected` message to return the selected column names and then insert the columns in a Bivariate plot.

Note: Method two and method three of extracting values are alternate examples of the same method. The method, in both examples, uses a JSL callback from one of the controls in the window. The JSL callback captures a value from a display box in the window into a global variable that will be available after the window closes. In method two, the JSL callback is on a button box. In method three, the JSL callback is on a list box.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
xvar = .;
yvar = .;
win = New Window( "Return Values",
   <<Modal,
   // require the user to select two variables before clicking OK
   <<On Validate(
      Show( xvar, yvar );
         If( Is Missing( xvar ) | Is Missing( yvar ),
            0, // if xvar or yvar are missing, do nothing when OK is clicked
            1
      );
   ),
   Text Box( " Select two numeric columns. " ),
   H List Box(
      Text Box( " X, Factor " ),
      x = Col List Box(
         dt, // data table reference
         all, // display all columns from the data table
         xvar = (x << Get Selected)[1];
         // get the name of the selected column before the window closes
         Show( xvar );
      ),
      Text Box( "Y, Response" ),
      y = Col List Box(
         dt,
         all,
         yvar = (y << Get Selected)[1];
         Show( yvar );
      )
   )
);
xcol = Column( dt, xvar ); // get the columns
ycol = Column( dt, yvar );
```

```
dt << Bivariate( Y( ycol ), X( xcol ) ); // create a Bivariate plot
```

Constructors for New Windows

JSL supports both modal and non-modal windows.

- Modal windows must be closed before other windows can be used: clicking outside the modal window produces an error sound, and almost all script execution stops until the user closes the modal window.

- JMP reports and platform launch windows are examples of non-modal windows. Many non-modal windows can be open at once and you can switch between them freely. The Transparency dialog in a graph is an example of a modal window. After the Transparency dialog is opened, you must close it before continuing.

This section describes common display boxes that you can construct in a window. There are many other display boxes (such as axis boxes) in JMP that cannot be constructed. Display boxes that cannot be constructed are used by platforms in JMP. Even though they can't be constructed in JSL, they still have functions and messages that can be used from JSL.

Tip: In the Scripting Index (**Help** > **Scripting Index**), display boxes that cannot be constructed are marked by ◁ . Display boxes that you can construct are marked by ◻ .

Notes:

- The `Dialog()` function, which creates modal windows, is deprecated and will not work at all in future versions of JMP. Use either `New Window()` with the `Modal` message or `Column Dialog()` for column selection. See "Construct a Column Dialog" on page 611 and "Convert Deprecated Dialog to New Window" on page 617.

- In JMP, the following display boxes support only one direct child box: `Border Box()`, `Center Box()`, `If Box()`, `Mouse Box()`, `Scroll Box()`, and `Sheet Panel Box()`. To make a border box that appears to have multiple children, use a `V List Box()` or `H List Box()` as the only child and put multiple children in the `V List Box()` or `H List Box()`.

Container Display Boxes

Container display boxes enclose child display boxes.

Border Box

`Border Box()` adds space around the *displaybox* argument.

```
Border Box (Left( pix ), Right( pix ), Top( pix ), Bottom( pix ),
    Sides( int ), display box args);
```

Left, Right, Top, and Bottom add space around the *displaybox* argument. Sides draws a border around the box, as described in Table 11.2. Additional effects can also be applied to the borders using Sides, as described in Table 11.2. To add both an effect and a border, add the two numbers.

For example, the following example produces a text box with a border at the top and bottom (Draw Border value of 5) colored blue, the default color (Effect value of 16):

```
win = New Window( "Borders",
  bb = Border Box( Sides( 5 ), // top and bottom border
    Text Box( "Hello World!" )
  )
);
bb << Set Color( "red" ); // set the border color to red
```

The bb variable is assigned to the border box so that the color can be set on the last line.

Note: Border boxes support only one display box argument. To make a border box that appears to have multiple children, use a V List Box() or H List Box() as the only child and put multiple children in the V List Box() or H List Box().

In the Sides() function, add the following values to get the borders that you want. For example, "5" draws a top and a bottom border.

1: top

2: left

4: bottom

8: right

Border Box() provides the additional effects described in Table 11.2.

Table 11.2 Border Box Sides() Additional Effects

Add to Number Above	Additional Effect
16	Make the border lines the same color as the operating system's highlight color. Not recommended because this color is different on different computers.
32	Fill the border box with white before drawing the border box's child.

Table 11.2 Border Box Sides() Additional Effects *(Continued)*

Add to Number Above	Additional Effect
64	Fill the border box with the background color before drawing the border box's child. This option makes a difference only if the border box's parent draws something behind the border box first.

Col List Box

`Col List Box()` returns a display box that shows a list of current data table columns.

```
Col List Box( <Data Table( <name> ), <all>|<character>|<numeric>,
    <Width( n )>, <Max Selected( n )>, <NLines( n )>, <Max Items( n )), <Min
    Items( n )>, <On Change( expression )>, <script> );
```

Note: To add a transform column to a custom display box, use `Filter Col Selector()` instead of `Col List Box()`. See "Filter Col Selector" on page 546.

`All` specifies that all columns in the current data table should be included. `Width` is measured in pixels. `Max Selected` is the maximum number of items that might be selected in the list box. `NLines` is the number of lines displayed in the box. `Max Items` is the maximum number of items displayed in col list box. `Min Items` is the minimum number of items displayed in the col list box. `On Change` is the script that runs when the selection changes.

You can send a `Get Items` message to a col list box to retrieve a list of all columns selected. Here is an example script showing `Get Items` in use. See the Scripting Index in the Help menu for other messages and examples.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Get Items Demonstration",
    H List Box(
        chooseme =

        // Col List Box shows all columns, 100 pixels wide with 6 lines
        Col List Box( "All", width( 100 ), NLines( 6 ) ),

        /* line up box arranges button boxes in 1 column
        with 2 pixels of space around the boxes*/
        Line Up Box( N Col( 1 ), Spacing( 2 ),

            Button Box( "Add Column >>",
```

```
                    /* append the selected column to previously
                    selected columns */
                    listcols << Append( chooseme << Get Selected );

                    // get a list of selected columns from the col list box
                        Chosen Columns = listcols << Get Items;
                            ),
                Button Box( "<< Remove Column",
                    listcols << Remove Selected; // remove the selected column
                    Chosen Columns = listcols << Get Items; // get the selected column
                ),
            ),
                // create another col list box with no columns inside
                listcols = Col List Box( width( 100 ), NLines( 6 ) ),
        ),
        Text Box( " " ),

        /* at the bottom of the h list box,
        show which columns Get Items returned */
        stuff = Global Box( Chosen Columns )

    );
```

The All argument for Col List Box() includes all data table columns in the display box. Because all columns are required, you cannot remove selected rows using Remove Selected or Remove All. Instead, create an empty column list box and append all columns. Then the columns that you select can be removed.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
colNames = dt << Get Column Names; // get a list of column names

New Window( "Example",
// create Col List Box with all the column names from the data table
myBox = Col List Box( width( 200 ), nlines( 10 ) ) );
myBox << Append( colNames );

myBox << Set Selected( 3 ); // select one or all of the entries in myBox
myBox << Remove Selected(); // remove selected entries in myBox
```

Note: To show only the character or numeric columns from the data table in the col list box, send Set Data Type(numeric|character) to the col list box.
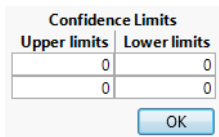
Col Span Box

`Col Span Box()` creates spanned columns headers inside a table box. The top column header spans two child column headers.

```
Col Span Box( "title", display box args);
```

In the following script, the Confidence Limits header spans the Upper limits and Lower limits columns below it.

```
win = New Window( "Col Span Box",
    <<Modal,
    Table Box(
        Col Span Box(
            "Confidence Limits",
            neb = Number Col Edit Box( "Upper limits", [0, 0] ),
            Number Col Edit Box( "Lower limits", [0, 0] )
        )
    )
);
```

Figure 11.9 Confidence Limits

Filter Col Selector

`Filter Col Selector()` creates a display box that shows a list of items, for example, columns in a data table. The display box is similar to `Col List Box()` except that `Filter Col Selector()` enables the user to filter columns by name, modeling type, data type, and so on. And `Filter Col Selector()` supports transform columns; `Col List Box()` does not.

```
Filter Col Selector( <Data Table( name )>, <Width( pixels )>, <Nlines( n )>,
    <script>, <On Change( expr )>, display box args, ... );
```

Here's an example that shows how to use `Get Items` to display a list of selected columns from the filter col selector.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Filter Col Selector Example",
    H List Box(
        chooseme =

        // filter col selector shows all columns, 100 pixels wide with 6 lines
        Filter Col Selector( width( 100 ), NLines( 6 ) ),
```
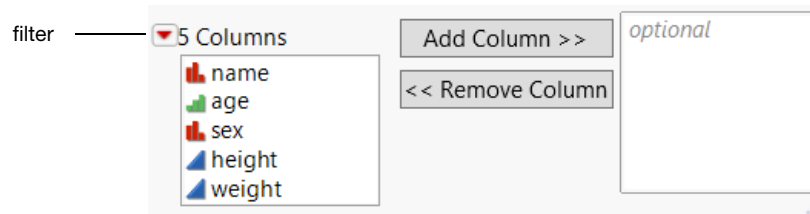
```
        /* line up box arranges button boxes in 1 column
        with 2 pixels of space around the boxes */
        Line Up Box( N Col( 1 ), Spacing( 2 ),
                Button Box( "Add Column >>",

            /* append the selected column to previously
            selected columns */
            listcols << Append( chooseme << Get Selected );

                // get a list of selected columns from the filter col selector
                Chosen Columns = listcols << Get Items;
                    ),
            Button Box( "<< Remove Column",
                listcols << Remove Selected; // remove the selected column
                Chosen Columns = listcols << Get Items; // get the selected column
            ),
        ),
            // create another col list box with no columns inside
            listcols = Col List Box( width( 100 ), NLines( 6 ) ),
    )
);
```

Figure 11.10 Filter Col Selector

H List Box and V List Box

H List Box() aligns boxes horizontally.

```
    H List Box( <Align( center|bottom )>, display box args, ...);
```

Align specifies center or bottom alignment of the contents in the display box. The contents are top aligned by default.

The following script aligns the check boxes horizontally:

```
    win = New Window( "H List Box",
        <<Modal,
        H List Box(
            kb1 = Check Box( "a" ),
            kb2 = Check Box( "b" ),
            kb3 = Check Box( "c" )
```

```
        ),
    );
```

`V List Box()` aligns boxes vertically.

```
    V List Box( <Align( center|right )>, display box args, ...);
```

`Align` specifies left, right, or center alignment of the contents in the display box. The contents are left aligned by default.

```
    win = New Window( "V List Box",
        Text Box( "Select your favorite fruit." ),
        V List Box(
            Check Box( "apple" ),
            Check Box( "banana" ),
            Check Box( "orange" ),
        ),
    );
```

Note: The default maximum width is 180 pixels. To set the width, set the width of the display box that encloses the V List Box.

Suppose that you want to enable the user to add a transform column to custom dialog box. Here is an example of how to create a horizontal list box with an Add and Remove button. The user right-clicks a column to create a transform column.

```
    dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
    New Window( "Example",
        H List Box(
            ll1 = Filter Col Selector( ),
            Button Box( "Add", ll2 << Append( ll1 << Get Selected() ) ),
            ll2 = Col List Box( "Numeric", MaxItems( 1 ), nlines( 1 ) ),
            Button Box( "Remove", ll2 << Remove Selected )
        )
    );
```

Note: Transform columns use formulas or calculations to define the column values. Closing the window deletes any transform columns. See *Using JMP*.

If Box

`If Box()` creates a display box whose contents are conditionally displayed.

```
    If Box( 0|1, display box args );
```

The Boolean argument 1 displays the display boxes inside the if box. 0 does not display them.

The following script creates a splitter box with three scroll boxes. The second scroll box is enclosed in `If Box()`. Because `If Box()` is set to 1, the second scroll box appears in the window.

```
win = New Window( "If Box",
    H Splitter Box(
        Size( 500, 250 ),
        Scroll Box(),
        If Box( 1, Scroll Box() ), // if box sets the scroll box to show
        Scroll Box()
    )
);
```

When you set an if box to be hidden, then anything within the if box disappears from the report window. In the tree structure view, the hidden display box is no longer numbered. Hide the if box in this example, and the second scroll box is no longer numbered in the tree structure view. `Scroll Box(3)` becomes `Scroll Box(2)`. Change the argument for `If Box()` in this script from 1 to 0 and then send the `win << Show Tree Structure` message to see the structure view.

Note: If boxes support only one display box argument.

H Splitter Box and V Splitter Box

`H Splitter Box()` and `V Splitter Box()` create a display box that arranges the display boxes provided by the arguments in a horizontal or vertical layout (or panel). The splitter enables the user to interactively resize the panel.

```
H Splitter Box( <Size( h,v )>, display box args, <arguments> );
V Splitter Box( <Size( h,v )>, display box args, <arguments> );
```

`Size( h, v )` specifies the size of the splitter box in pixels. Inner display boxes are proportionately sized according to the width and height of the splitter box. For a list of arguments, see the Scripting Index in the JMP Help menu.

The following example puts a Bivariate and Oneway graph inside an H splitter box. Splitter boxes are especially useful when the content is set to auto-stretching as shown in the example.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "H Splitter Box",
    H Splitter Box(
        Size( 900, 600 ),
        biv = dt << Bivariate( Y( :weight ), X( :height ), Fit Line(), Line
    Style( Dotted ) ),
        one = Oneway( Y( :height ), X( :sex ), "Means/Anova"n( 1 ), Mean
    Diamonds( 1 ) )
    )
);
```

```
/* make a reference to the report layers
set the x and y axes on the frame boxes to auto stretch
when the user drags the splitter */
(biv << Report)[FrameBox( 1 )] << Set Stretch( "Window", "Window" );
(one << Report)[FrameBox( 1 )] << Set Stretch( "Window", "Window" );
```

Line Up Box

Line Up Box() shows the *display box* arguments in *n* columns. You can specify optional spacing, in pixels, for the space between columns.

```
Line Up Box( NCol( n ), <Spacing( pixels )>, display box args, ... );
```

Spacing specifies the number of pixels around the box.

In the following example, the Distribution reports are lined up in three columns:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Line Up Box",
    Line Up Box( NCol( 3 ), // line up the Distributions in 3 columns
        dist = Distribution(
            Continuous Distribution( Column( :height ) ),
            By( :age )
        )
    )
);
```

Outline Box

Outline Box() creates a display box to hold other display boxes and show them in an outline view.

```
Outline Box("title", display box args, ... );
```

In the following example, each outline contains a Distribution:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Outline Box",
    obj1 = Outline Box( "Age and Weight",
        Distribution( Column( :age, :weight ) ),

    ),
    obj2 = Outline Box( "Weight and Height",
        Distribution( Column( :weight, :height ) )
    )
);
```

To close an outline by default, send the `Close( 1 )` message to the outline object.

```
obj2 << Close( 1 );
```

The first argument for an outline box is the title. You can specify the title in the script, or the title can be based on another component, such as the value of a slider box. In the following example, as the user adjusts the slider, the title of the outline box is updated:

```
sliderValue = .6;
win = New Window( "Outline Box",
  Panel Box( "Slider Box",
      tb = Outline Box( "Value: " || Char( sliderValue ) ),
      sb = Slider Box(
         0,
         1,
         sliderValue,
         tb << Set Title( "Value: " || Char( sliderValue ) )
      )
   )
);
```

To make your own red triangle menu within an outline box, send the `Set Menu Script` message to the outline box reference. The following example also uses the `Set Submenu` message to create nested menu items.

```
win = New Window( "Red Triangle Menu Items",
ob = Outline Box( "Outline Box" ) );
ob << Set Menu Script(
  {"A", "", "A1", Print( "A1" ), "A2", Print( "A2" ), "B", "",
  "B1", Print( "B1" ), "B2", Print( "B2" ), "B3",
  Print( "B3" ), "C", Print( "C" ), "Selected", Print( "Selected" )}
);
ob << Set Submenu( 1, 2 ); // menu A with 2 items in submenu A1 and A2
ob << Set Submenu( 4, 3 ); // menu B with 3 items in submenu B1, B2, and B3
```

To indicate whether an item is selected, unavailable, or available, include `Set Menu Item State`. The following expression shows a check mark beside the ninth menu item ("Selected") in the preceding script.

```
ob << Set Menu Item State( 9, 1 );
```

Specify the value -1 to disable the menu item or 0 to make it available.

Panel Box

`Panel Box()` encloses the *displaybox* argument in a labeled border.

```
Panel Box( "title", display box args );
```

In the following example, the panel box appears around the edge of the Distribution launch window:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Panel Box",
    H List Box( Panel Box( "My Distribution", Distribution() ) )
);
```

Display Element Boxes

Graph Box

Graph Box() constructs a display box that contains a frame box and axes.

```
Graph Box( "title", <X Scale( low, high ), <Y Scale( low, high )>, <Frame
    Size( h, v )>,< <XName( "name" )>, <YName( "name" )>, <SuppressAxes>,
    script );
```

X Scale() and Y Scale() specify the lower and upper limits of the x and y axes. Frame Size() specifies the size of the frame around the graph. XName and YName specify the name of the x and y axes.

The following example creates a simple graph:

```
win = New Window( "Graph Box",
    Graph Box(
        Frame Size( 300, 300 ),
        Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
        Pen Color( "Blue" );
        Line( [10 30 70], [88 22 44] );
    )
);
```

To remove tick marks, add two arguments to the axes.

```
win = New Window( "Graph Box",
    Graph Box(
        Frame Size( 300, 300 ),
        xaxis( // remove tick marks from the x axis
            Show Major Ticks( false ),
            Show Minor Ticks( false ),
            Show Labels( false )
        ),
        yaxis( // remove tick marks from the y axis
            Show Major Ticks( false ),
            Show Minor Ticks( false ),
            Show Labels( false )
        ),
        Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
```

```
        Pen Color( "Blue" );
        Line( [10 30 70], [88 22 44] );
    )
);
```

JMP provides a robust set of features to make your graphs interactive (such as dragging objects or capturing mouse clicks). The "Scripting Graphs" chapter on page 631 describes graph box options in detail.

H Sheet Box and V Sheet Box

Sheet Box() lets you create a grid of plots. H Sheet Box() and V Sheet Box() contain display boxes and arrange them in columns and rows.

```
H Sheet Box( <<Hold( report ), display box args );
V Sheet Box( <<Hold( report ), display box args );
```

The general approach is to first consider which display boxes you want and in which arrangement. Create either an H or V Sheet Box and send it a Hold message for each graph to specify which element the sheet box should hold. Finally, create interior H or V Sheet Boxes and tell each one which graph the sheet box should hold.

Here is an example of creating a sheet with four plots: a Bivariate plot, a Distribution, a Treemap, and a Bubble Plot.

First, open the data table and create a new window.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
nw = New Window( "Sheet Box Example",
```

Use a V Sheet Box to organize the window into two columns.

```
    V Sheet Box(
```

Send the V Sheet Box() four Hold messages, one for each graph. The order matters.

```
        <<Hold(
          dt << Bivariate( // graph 1
            Y( :weight ),
            X( :height ),
            Fit Line()
          )
        ),
        <<Hold(
          Distribution( // graph 2
            Continuous Distribution(
              Column( :height ),
              Horizontal Layout( 1 ),
              Outlier Box Plot( 0 )
            )
```

```
        )
    ),
    <<Hold( Treemap( Categories( :age ) ) ), // graph 3
    <<Hold(
        Bubble Plot( // graph 4
            X( :height ),
            Y( :weight ),
            Sizes( :age ),
            Coloring( :sex ),
            Circle Size( 6.226 ),
            All Labels( 0 )
        )
    ),
```

Finally, add two H Sheet Boxes to the V Sheet Box and tell each one which graph it should hold. Each H Sheet Box holds two side-by-side graphs. They are held by a V Sheet Box, so the H Sheet Boxes are displayed vertically.

```
    H Sheet Box(
        Sheet Part(
            "",
```

Sheet Part displays a previously defined graph, held by Excerpt Box. The first argument is the number of the graph, determined by the order in which you defined the graphs. The first H Sheet Box contains the Bivariate graph on the left and the Distribution on the right. {Picture Box(1)} designates which picture box from the report to display. Typically, use 1, the first picture box in the report.

```
            Excerpt Box( 1, {Picture Box( 1 )} ) // excerpt box 1
        ),
        Sheet Part(
            "Distribution of height",
            Excerpt Box( 2, {Picture Box( 1 )} ) // excerpt box 2
        )
    ),
    H Sheet Box(
        Sheet Part(
            "",
            Excerpt Box( 3, {Picture Box( 1 )} ) // excerpt box 3
        ),
        Sheet Part(
            "My Title Here",
            Excerpt Box( 4, {Picture Box( 1 )} ) // excerpt box 4
        )
    )
  )
);
```

The Sheet Part() title is required. If you include an empty string as the title, the sheet part's title is the default report title, for example, "Bivariate Fit of weight By height".

Journal Box

The Journal Box() constructs a display box from instructions that are stored in a journal.

```
box = Journal Box("journal text");
```

where *"journal text"* is text that has been extracted from a journal file.

Since journal text has lots of rules about which boxes can be with other boxes, we recommend that you get journal text by highlighting an area of the report, selecting Edit > Journal, and save it as a journal. Open the journal in a text editor, and then paste it into your script as the Journal Box() argument.

We highly recommend that you use the "\[...]\" quoting mechanism so that you do not have to escape double quotes within the journal text. This excerpt shows how to escape the journal text:

```
win = New Window( "Mosaic Plot",
    Journal Box(
        "\[ // beginning of quoting mechanism
    ...display box arguments...
    ]\"
    ) // end of quoting mechanism
);
```

Another way to get journal text is to send the Get Journal message to a display box. For example, run the following script and copy the journal text from the log into the Journal Box() function:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :weight ), x( :height ) );
rbiv = biv << Report;
Print( rbiv << Get Journal ); // print the journal to the log
```

Picture Object

JSL has a Picture Object, which stores pictures of JMP output or formulas. You can take a picture of anything in a display box or create a picture of a text formula as it would look in the Formula Editor.

To create picture data, send a Get Picture message to a *display box*.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :weight ), x( :height ) );
rbiv = biv << Report;
rbiv << Get Picture;
```

The `Expr As Picture()` function evaluates its argument and creates a picture of the expression, using the same formatting mechanism as the Formula Editor. If you have a literal expression as the argument, remember to enclose it in `Expr()` so that JMP just takes a picture of the result. The expression then is not evaluated.

```
win = New Window( "Formula",
  Expr As Picture( Expr( a + b * c + r / d + Exp( x ) ) )
);
```

After you have the picture, there are two ways of using it:

- Add the picture to a new display tree using a display box constructor.

- Write the picture to a file using `Save Picture()`.

  ```
  picture << Save Picture( "path", type );
  ```

  where *type* can be EMF (Windows), JPEG or JPG, GIF, PDF, or PNG.

On Windows, the Windows Specific preferences determine the resolution (or DPI), or you can run this expression:

```
Pref( Save Image DPI( <number> ) );
```

On macOS, the operating system determines the DPI.

Input Selector Display Boxes

Input selector display boxes enable users to enter information and interact with the display box. The display box script includes a script that runs when the user interacts with the display box. See "Set Function and Set Script" on page 588 for more information about writing the scripts.

Button Box

`Button Box()` draws a button that contains text.

```
Button Box("text", <script>);
```

The following example creates a new window with an OK button and a Cancel button:

```
win = New Window( "Set the Value",
  <<Modal,
  Text Box( "Set this value" ),
  variablebox = Number Edit Box( 42 ),
  Button Box( "OK" ),
  Button Box( "Cancel" ) );
```

Note that a modal window must have at least one button. If you omit the `Button Box()` from your script, an OK button is automatically included. To omit the OK button, set the visibility to collapse.

```
win = New Window( "No OK Button",
    <<Modal,
    b = Button Box( "Close Window", b << Close Window ),
    bb = Button Box( "OK", <<Visibility( "collapse" ) )
);
```

You can add a tooltip for a button by sending a Set Tip message to it. The following script creates a button named "Submit". A tooltip that contains "Send my information." appears when you hover over the button.

```
Button Box( "Submit", << Set Tip( "Send my information." ) );
```

You can also send the message Open Next Outline as a script command, which causes the next outline box to open. If you are sending more than one message to the button box, this must be the first command listed. When the user clicks the next button in this example, the second outline box opens.

```
win = New Window( "Open Next Outline",
    Outline Box( "First",
        ex = Button Box( "Next", ex << Open Next Outline )
    ),
    Outline Box( "Second", Text Box( "Last" ), << Close )
);
```

Using the Set Menu Items message, you can create a button that contains a menu:

```
win = New Window( "Set and Get Menu Choice",
    bb = Button Box( "Select a dessert",

    // get menu choices when the button is clicked
        choice = bb << Get Menu Choice;
        Show( choice );
    )
);
bb << Set Menu Items( {"fruit", "no thank you", "-", "chocolate bar", "ice
    cream"} );
```

The "-" in the preceding script creates a menu separator. The separator counts as an item in the list. If you select chocolate bar from the menu, 4 is returned, not 3.

To indicate whether an item is selected, unavailable, or available, include Set Menu Item State. The following example makes the fifth menu item ("ice cream") in the preceding script unavailable:

```
bb << Set Menu Item State( 5, -1 );
```

Specify 0 to make the menu item available or 1 to show a check mark before the menu item.

Note that, after you click the button, the button box doesn't know where the script is stored because the button is *disconnected* from the script. For example, relative paths don't work when executed from a button box because of this disconnection. An Open() script must evaluate the current directory and concatenate it to the relative file path to construct an absolute path.

```
nw = New Window( "Open File Example",
    Button Box( "Open",
        Open(
            Eval( Get Path Variable( "$SAMPLE_DATA" ) ) ||
            "Quality Control/Coating.jmp"
        )
    )
);
```

In the following example, the path is interpreted as an absolute path beginning with the root.

```
nw = New Window( "Example",
Button Box( "Open", Open( "/MyData/Big Class.jmp" ) ) );
```

Note: Line-break characters are ignored in button boxes.

Check Box

Check Box() constructs a display box to show one or more check boxes. Any number of the check box items can be selected simultaneously.

```
Check Box ({"item 1", "item 2", ...}, <script>);
```

The following example shows how to get the user's selection by sending the Get Selected message to the check box. Each selection is printed to the log.

```
win = New Window( "Check Box",
    cb = Check Box(
        {"apple", "banana", "orange"},
        scb = cb << Get Selected();
        Show( scb );
    )
);
    scb = {"banana", "orange"};
```

The following example shows one way to enable the user to click OK to close the window.

```
win = New Window( "Check Box",
    <<Modal,
    <<On Close(
        flagbox1 = ckbox <<Get( 1 );
        flagbox2 = ckbox <<Get( 2 );
        1; // On Close returns non-zero, so the window closes.
    ),
```

```
    ckbox = Check Box( {"Check Box 1", "Check Box 2"} ),
    ckbox << Set( 1, 1 ),
    ckbox << Set( 2, 0 )
);
```

Combo Box

Combo Box() creates a drop-down list.

```
    Combo Box( {"item 1", "item 2", ...}, <script> );
```

"item1" and "item2" are strings in the drop-down list.

The Editable argument allows the user to enter text in the combo box. The following example creates a new window and an editable combo box with the list items "one", "two", and "three". The script prints the selected item name and index number to the log.

```
    win = New Window( "Combo Box",
        cb = Combo Box(
            {"One", "Two", "Three"},
            Editable,
            selection = cb << GetSelected();
            Print( "Selected: " || selection );
            Print( "Index: " || Char( cb << Get() ) );
        )
    );
```

Note that closing the window for an editable combo box can produce unexpected results. On Windows, the Print() script runs after you select an item from the list or enter the item name. On macOS, the Print() script runs after you select an item from the list or enter the item name and then press Return.

For cross-platform compatibility, include !Is Empty to test for an existing combo box before running the script.

```
    win = New Window( "Combo Box",
        cb = Combo Box(
            {"One", "Two", "Three"},
            Editable,
            If( !Is Empty( cb ),
                selection = cb << GetSelected();
                Print( "Selected: " || selection );
                Print( "Index: " || Char( cb << Get() ) );
            )
        )
    );
```

To make an item selectable or unselectable, send the <<Enable Item message to Combo Box().

```
cb << Enable Item( 2, 0 ) // makes the second item unselectable
```

Note: If the editable combo box has no empty strings, nothing is selected by default. If the editable combo box has one or more empty strings, the first empty string is selected by default

Global Box

Global Box shows the name and current value of a JSL global variable.

```
Global Box( name );
```

The user can assign a new value to the global variable by editing the value directly in the window and pressing Enter or Return to commit the change. Global boxes automatically update the displayed value of a variable when the variable changes.

```
ex = Sqrt( 4 );
win = New Window( "Global", Global Box( ex ) );
```

The preceding script creates a new window named "Global" that displays the following result:

   ex=2

The following example shows the value of the myGlobal variable on the graph:

```
myGlobal = 6;
    win = New Window( "Global Box",
        V List Box(
            gr = Graph Box(
                Frame Size( 300, 300 ),
                X Scale( 0, 10 ),
                Y Scale( 0, 10 ),
                Y Function( x, x ),
                Text( {5, 5}, "My global is ", myGlobal )
            ),
            Global Box( myGlobal ) // value of myGlobal
        )
    );
```

Note: When you are using Global Box(), remember that every time the variable is changed, JMP remeasures, reshows, and updates the window. Depending on the number of Global Box() objects, changes to the variable can affect the refresh rate of the window. JMP recommends avoiding numerous global boxes in deliverable scripts. Global boxes can be replaced with text boxes that the script can manually update when necessary.

List Box

List Box() creates a display box that shows a list box of items that can be selected.

```
List Box({"item 1", "item 2", ...}, <Width( n )>, <MaxSelected( n )>,
   <NLines( n )>, <script>);
```

Item names are case sensitive by default. Width is measured in pixels. MaxSelected is the maximum number of items that might be selected in the list box. NLines is the number of lines to display in the box, with a default value of 3.

Add items to a list box by either using Append or Insert:

```
win = New Window( "test", lb = List Box( {"a", "e"} ) );
lb << Append( {"f", "g"} ); // result is a, e, f, g
lb << Insert( {"b", "c", "a", "d"}, 1 ); // result is a, b, c, d, e, f, g
```

Append always adds the list to the end of the list box. Insert adds the list after the position specified.

You can have two mutually exclusive list boxes, so that the item you select in one box is deselected when you select an item in the other box. Send the Clear Selection message to deselect the item.

```
win = New Window( "Each Box Clears the Other Box",
   window:la = List Box(
       {"broccoli", "pepper", "spinach"},
       <<Set Script( window:lb << Clear Selection )
   ),
   window:lb = List Box(
       {"avocado", "pumpkin", "tomato"},
       <<Set Script( window:la << Clear Selection )
   )
);
```

A list box can also contain an image. In the following example, an image from the user's computer appears in the "first" list item. The JMP nominal icon appears in the "second" list item.

```
win = New Window( "List Box with Image",
    List Box(
        {{"first", "$SAMPLE_IMAGES/pi.gif"}, {"second", "nominal"}},
        width( 200 ),
    )
);
```

To set the width of the list box, use the width argument.

```
listA = {"a", "b", "c",
"This is a very, very long name like Mary Ann Johnston."};
New Window( "Example",
```

```
   // set the width to the length of the string
      V List Box( listBoxA = List Box( listA, width( 0 ) ) ) );
```

To set the font of the list box, use the Set Base Font message.

```
New Window( "Example",
   fontobj = lb = List Box(
      {"First Item", "Second Item", "Third Item"},
      width( 200 ),
      max selected( 2 ),
      nlines( 6 )
   )
);
Wait( 2 );
fontobj << Set Base Font( "Title" );
```

Set Base Font also works with Bar Seg, Col List Box(), Contour Seg, Filter Col
Selector, Hier Box(), Number Edit Box(), Outline Box(), Poly Seg, Text Box(), Text
Edit Box(), and Text Seg.

Mouse Box

Mouse Box() creates a box that can make JSL callbacks for dragging and dropping, marking,
or clicking and tracking mouse actions.

```
Mouse Box( display box args, messages );
```

The following example is from the DragDrop.jsl sample script in the Samples/Scripts folder.
One list box contains "a", "b", and "c". The other list box contains "d", "e", "f". You can drag
a letter from one list box to the other list box.

Note: Mouse boxes support only one display box argument.

```
win = New Window( "Drag and drop between list boxes",
   H List Box(
      MouseBox(
         one = List Box( {"a", "b", "c"} ),

         // if the mouse is over the preceding list box, drop on the other box
         <<setTrack(
            (two << parent) << setDropEnable( 1 );
            (one << parent) << setDropEnable( 0 );
         ),

         // watch the mouse movement (not clicks)
         <<setTrackEnable( 1 ),
```

```
            // when a drop happens...
            <<setDropCommit(
               Function( {this, clickPt, text},
               // find out what is selected in the other box
                  x = two << getSelected;
                  If( Words( text, "\!t" ) == x,
                     two << removeSelected; // and take it...
                     one << append( x ); // and add it to me
                  ,
                     Print( "Drag from the other box please." )
                  );
               )
            )
         ),
         Spacer Box( size( 20, 20 ) ),

         /* same comments as above with reversed logic
         for current box versus other box */
         MouseBox(
            two = List Box( {"d", "e", "f"} ),
            <<setTrack(
               (one << parent) << setDropEnable( 1 );
               (two << parent) << setDropEnable( 0 );
            ),
            <<setTrackEnable( 1 ),
            <<setDropCommit(
               Function( {this, clickPt, text},
                  x = one << getSelected;
                  If( Words( text, "\!t" ) == x,
                     one << removeSelected;
                     two << append( x );
                  ,
                     Print( "Drag from the other box please." )
                  );
               )
            )
         )
      )
   )
);
```

Number Col Edit Box

Number Col Edit Box() creates a display box that shows editable numbers in a column. The numeric entries can be in list or matrix form.

```
Number Col Edit Box("title", numbers);
```

The following script creates number edit boxes in which the user enters values. The Return Result message extracts the values.

```
x = y = z = 0;
win = New Window( "Number Col Edit Box",
    <<Modal,
    <<Return Result,
    Table Box(
        neb = Number Col Edit Box( "values", {x, y, z} )
    )
);
    {neb = {2, 4, 6}, Button( 1 )}
```

Number Edit Box

Number Edit Box() creates an editable number box that initially contains the value argument. You can also set the width of the box in characters.

```
Number Edit Box( initValue, <width>);
```

The following script shows an editable box that initially contains the number "42". The number that the user types is printed to the log.

```
win = New Window( "Set a Value in Number Edit Box",
    <<Modal,
    <<Return Result,
    Text Box( "Set this value" ),
    variablebox = Number Edit Box( 42 ),
    Button Box( "OK" ),
    Button Box( "Cancel" )
);
Write( win["variablebox"] ); // create a subscript to the variablebox variable
    33 // the user typed "33" in the number edit box
```

The following script shows how to make the initial value a date, and enables the user to choose a different date.

```
New Window( "Date/Time Selector",
    <<Modal,
    neb = Number Edit Box(
        Today(),
        23,
        <<Set Format( Format( "m/d/y h:m:s", 23, 0 ) )
    )
);
```

Popup Box

`Popup Box()` creates a red triangle menu that contains menu item commands.

```
Popup Box( {"command1", script1, "command2", script2, ...} );
```

The following example stores a command list in a variable and then uses `Popup Box()` to display the items:

```
List = {
"command1", Print( "command1" ),
"command2", Print( "command2" ),
"command3", Print( "command3" ),
"command4", Print( "command4" ),
"", Empty(), // make a separator line
"command5", Print( "command5" ),
"commandThrow", Throw( "commandThrow1" ),
"commandError", Sqrt( 1, 2, 3 ),
"commandEnd", Print( "commandEnd" )};

win = New Window( "Example",
  Text Box( "Popup Test" ),
   Popup Box( List );
);
```

Figure 11.11 Red Triangle Menu

Note that you can disable and re-enable the menu using the `Enable( Boolean )` message. An argument of 1 turns the menu on, and an argument of 0 turns the menu off. Using the previous example, assign the popup box to a variable and then send messages to it:

```
win = New Window( "Example",
  Text Box( "Popup Test" ),
   mymenu = Popup Box( List );
);
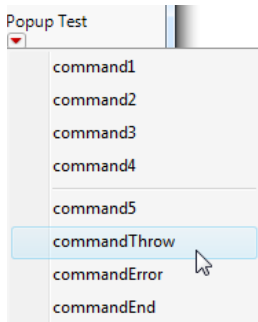
mymenu << Enable( 0 ); // disabled menu
```

You can also add a popup menu to an outline box. See "Outline Box" on page 550 for an example.

Radio Box

Radio Box() creates a list of radio buttons.

```
Radio Box({"item 1", "item 2", ...}, <script>);
```

Only one item in the radio box can be selected at any time.

The following script creates "A", "B", and "C" radio buttons and selects the second box.

```
win = New Window( "Radio Box",
  V List Box(
    rb = Radio Box( {"A", "B", "C"}, <<Set( 2 ) ),
  )
);
```

"New Window Example" on page 617 also provides an example of creating radio boxes.

Slider Box

Slider Box() creates a slider control for picking any value for the variable, within the minimum and maximum range that you specify.

```
Slider Box( min, max, global variable, script, <Set Width( n )>, <Rescale
  Slider( min, max )> );
```

Set Width specifies the width of the slider box. Rescale Slider specifies the minimum and maximum values.

When the slider is moved, the value given by the current position of the slider is assigned to the global variable. Thus, Slider Box() is another way to parameterize a graph.

```
ex = .5;
win = New Window( "Slider Box",
  tb = Text Box( "Value: " || Char( ex ) ),
  sb = Slider Box(
    0,
    1,
    ex,
    tb << Set Text( "Value: " || Char( ex ) )
  )
);
sb << Set Width( 100 ) << Rescale Slider( 0, .8 );
```

The following script creates multiple slider boxes without using a unique global variable to hold the current value for each one:

```
sb1 = Slider Box( 1, 10 );
```

```
// create a slider box with only the min and max value
sb2 = Slider Box( -10, 10 );
sb1 << Set Function( // set the script or the function
  Function( {this},
     {},
     Show( this << Get, sb2 << Get )
  )
);
sb2 << Set Script( Show( sb2 << Get, sb1 << Get ) );
New Window( "Values", sb1, sb2 ); // place the slider boxes in a window
```

As you move the slider, the value is printed to the log.

See for another example of slider boxes.

Spacer Box

Spacer Box() creates space between display boxes.

```
Spacer Box(<size(horizontal_pixels,vertical_pixels)>, <color(color)>)
```

The following script places a 450 pixel wide and 15 pixel tall blue spacer box between the two outline boxes.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Spacer Box",
  V List Box(
    Outline Box( "Bivariate",
       biv = dt << Run Script( "Bivariate" ),
       // Color string consists of percentages of RGB values
       Spacer Box( Size( 450, 15 ), Color( {.53, .80, .98} ), )
    ),
    Outline Box( "Distribution",
       V List Box( dist = dt << Run Script( "Distribution" ), )
    )
  )
);
```

Figure 11.12 Spacer Box Example

String Col Edit Box

`String Col Edit Box()` creates a column named title in a table containing the string items listed. The string boxes are editable.

```
String Col Edit Box( "title", {strings} );
```

The following script creates a column of editable text boxes, preceded by the title "Names":

```
a = b = c = "";
win = New Window( "String Col Edit Box",
    <<Modal,
    <<Return Result,
```

```
      Table Box(
         seb =
         String Col Edit Box(
            "Names",
            {a, b, c}
            )
         )
   );
      {seb = {"apple", "orange", "banana"}, Button( 1 )}
```

Tab Box and Tab Page Box

Tab Page Box() organizes the title and contents of the page in one display box. When Tab Page Box() is inside Tab Box(), a tabbed window with multiple tabbed pages appears.

```
Tab Box( Tab Page Box( Title( "page title 1" ), <options>, contents of
   page 1), Tab Page Box( "page title 2", <options>, contents of page 2),
   ...);
```

The following script creates three tab boxes:

```
win = New Window( "Tab Box",
   Tab Box(
      Tab Page Box(
         Title( "First page" ),
         Text Box( "first line of first page" ),
         Text Box( "second line of first page" )
      ),
      Tab Page Box(
         Title( "Second page" ),
         Text Box( "first line of second page" ),
         Text Box( "second line of second page" )
      ),
      Tab Page Box(
         Title( "Third page" ),
         Text Box( "first line of third page" ),
         Text Box( "second line of third page" )
      )
   )
);
```

Figure 11.13 Tab Boxes

You can specify which tab should be selected by sending the `Set Selected(`*n*`)` message, where *n* is the tab number, to the tab box object.

The `Set Style` message lets you select the visual appearance of the tab box. The default value is `tab`. Other options include the following:

- `combo` creates a combo box.

- `outline` creates an outline node.

- `vertical spread` displays the tab title vertically.

- `horizontal spread` displays the tab title horizontally.

- `minimize size` bases the tab style on the width of the tab title. In the following example, the first tab title is fairly long:

```
win = New Window( "Tab Box",
  tb = Tab Box(
    Tab Page Box(
      Title( "First page of my tab box" ),
      Text Box( "first line of first page" ),
      Text Box( "second line of first page" )
    ),
    Tab Page Box(
      Title( "Second page" ),
      Text Box( "first line of second page" ),
      Text Box( "second line of second page" )
    ),
    Tab Page Box(
      Title( "Third page" ),
      Text Box( "first line of third page" ),
      Text Box( "second line of third page" )
    )
  )
);
tb << Set Style( "minimize size" );
```

Minimizing the size of the tab box converts the tab box into a combo box. Figure 11.14 compares the default and minimized tab boxes.

Figure 11.14 Default Tab Box (Left) and Minimized Tab Box (Right)

For `Tab Page Box()` examples, see "Examples of Creating a Dashboard from Two Reports" on page 596.

Tips:

- Unlike with other display boxes, you must define the title in the `Title()` function.

- The `Set Style` message works for both bare words and quoted words. However, you cannot assign the style to a variable and then pass the variable as the argument.

- You can combine `Tab Box()` with `Splitter Box()` to create docked tabs, which can be dragged to drop zones in a window to create a different layout. See "Examples of Creating a Dashboard from Two Reports" on page 596.

- To delete a tab, you can either send `Delete(index)` to the tab box, or you can send `Delete Box()` to a tab page box. Note that the entire display box (all tabs) is deleted, not just a single tab.

- Use the `Visibility("Collapsed")` message to hide a tab.

- When tabs in a tab list are not wide enough for the titles, send the `Set Overflow Enabled(1)` message to `Tab Box()`. A triangle symbol is shown to the right of a tab list, and the tab titles are truncated using ellipses.

- When `Tab Page Box()` is created or dragged outside a `Tab Box()`, it is a stand-alone container. The title appears in a shaded box at the top of the page, similar to the title in `Sheet Part()`; the title does not appear in an interactive tab.

- See "Write Tab Box and Tab Page Box Scripts" on page 629 for more information about the differences between `Tab Box()` and `Tab Page Box()`.

- In Bubble Plot and Graph Builder scripts, set the `Fit to Window` message to `"On"` to resize the window when you resize the graph in `Tab Box()`.

- To close a tab, first make a tab closeable by sending `Closeable(1)` to the tab page box and then send the `Close` message.

- Closeable tabs can be closed with the middle mouse button.

Text Box

`Text Box()` creates a non-editable text box. Text boxes are frequently used as labels of other controls.

```
Text Box( "text" );
```

You can format the text with HTML tags. For example, the following script formats text in bold:

```
win = New Window( "Formatted Text",
    Text Box( "This is <b>bold</b> text.",
    <<Markup) );
```

Make sure you close nested tags correctly and escape the quotes as shown here:

```
"This is <font color=\!"Blue\!" face=\!"Lucida Sans Unicode\!"
    size=\!"24\!">blue</font> text. This is a <background
```

```
color=\!"yellow\!">yellow</background> background. This is <font
color=\!"Blue\!"><background color=\!"yellow\!">blue text on a yellow
background</background></font>."
```

See "Example of Specifying Text and Background Colors" on page 608 for another example.

Note: Rotated text boxes support Markup text except for larger text boxes with multiple lines, multiple formats or justifications applied.

Text Edit Box

Text Edit Box() creates an editable text box.

```
Text Edit Box ( "text" );
```

You can add a script to a text edit box by sending it a script message. This is usually most convenient at the time the box is created. Simply add the script message as the last argument.

The following example prints "Changed" in the log each time the text edit box is changed:

```
win = New Window( "Text Edit Box",
   Text Edit Box( "Change Me", <<Script( Print( "Changed" ) ) )
);
```

Assign a reference to the text edit box object to access its contents. The following example prints the value of the text edit box to the log each time the box changes:

```
win = New Window( "Text Edit Box",
   teb = Text Edit Box( "Change Me",
      <<Script( Print( teb << Get Text ) )
   )
);
```

Text edit boxes are empty. However, you can specify placeholder text as a visual cue for the user to enter a value in the box. The placeholder text is only a hint and does not affect the value of the text field.

The following example produces the text edit box shown in Figure 11.15. The Hint option returns "mm/dd/yyyy" and formats it as light gray text.

```
win = New Window( "Text Edit Box",
   Text Edit Box( "Current Date" ),
   Text Edit Box( "", Hint( "mm/dd/yyyy" ) )
);
```

Figure 11.15 Text Edit Box with Placeholder Text

Tip: To omit the border around the editable box and the white background color, send the `White Box Style( 0 )` message to the text edit box.

If you need to use a text edit box for your user to enter passwords, you can apply a style to the text edit box to replace all characters entered with an asterisk. For example, the following example creates a text edit box whose value is the string "a". When a user types a new string into the text edit box, each character is displayed as an asterisk, and the message "Changed!" is printed to the log.

```
win = New Window( "Text Edit Box",
    teb = Text Edit Box( "a",
        Password Style( 1 ),
        Set Script( Print( "Changed!" ) )
    )
);
```

You can also send a text edit box a message to either use password style or to stop using password style.

```
q << Password Style( 1 ); // set the text edit box to password style
q << Password Style( 0 ); // set the text edit box to standard style
```

Filtering Display Boxes

`Data Filter Source Box()` and `Data Filter Context Box()` work together to filter data in multiple graphs. `Data Filter Source Box()` defines which graph is the "source" of the selection filter. When data are selected in the source graph, graphs in the common `Data Filter Context Box()` are updated.

Data Filter Context Box

`Data Filter Context Box()` creates a display box that enables you to filter data in multiple graphs. The function determines the number of graphs that the filter applies to, for example, all graphs in a report.

```
Data Filter Context Box( display box args );
```

In the following example, the `H List Box()` that contains the graphs is wrapped in `Data Filter Context Box()`. After you run the script, select values in the filter to see both graphs update.

```
dt = Open( "$SAMPLE_DATA/Hollywood Movies.jmp" );
win = New Window( "Shared Local Filter",
    Data Filter Context Box(

    // enclose the h list box that contains distribution and graph builder
        H List Box(
```

```
            dt << Data Filter( Local ), // add the local data filter
            dt << Distribution(
               Weight( :Profitability ),
               Nominal Distribution( Column( :Lead Studio Name ) ),
               Nominal Distribution( Column( :Genre ) ),
               Histograms Only
            ),
            dt << Graph Builder(
               Variables(
                  X( :Genre ),
                  Y( :Domestic Gross ),
                  Y( :Foreign Gross, Position( 1 ) )
               ),
               Show Control Panel( 0 ),
               Elements(
                  Bar(
                     X,
                     Y( 1 ),
                     Y( 2 ),
                     Legend( 2 ),
                     Bar Style( "Side by side" ),
                     Summary Statistic( "Mean" )
                  ),
                  Frequencies( 0 ),
               )
            )
         )
      )
   )
);
```

Data Filter Source Box

Data Filter Source Box() and Data Filter Context Box() work together to filter data
in multiple graphs. Data Filter Source Box() defines which graph is the "source" of the
selection filter. When data are selected in the source graph, graphs in the common Data
Filter Context Box() are updated.

```
   Data Filter Source Box( display box args );
```

The following script lets you select data in Distribution graphs, and only that data is shown in
the Graph Builder graph:

```
   dt = Open( "$SAMPLE_DATA/Hollywood Movies.jmp" );
   win = New Window( "Shared Local Filter",
      Data Filter Context Box(

      // enclose data filter source box and the graph builder platform
         H List Box(
```

```
                     Data Filter Source Box(
                         dt << Distribution( // use distribution as the source graph
                             Weight( :Profitability ),
                             Nominal Distribution( Column( :Lead Studio Name ) ),
                             Nominal Distribution( Column( :Genre ) ),
                             Histograms Only
                         )
                     ),
                     dt << Graph Builder(
                         Variables(
                             X( :Genre ),
                             Y( :Domestic Gross ),
                             Y( :Foreign Gross, Position( 1 ) )
                         ),
                         Show Control Panel( 0 ),
                         Elements(
                             Bar(
                                 X,
                                 Y( 1 ),
                                 Y( 2 ),
                                 Legend( 2 ),
                                 Bar Style( "Side by side" ),
                                 Summary Statistic( "Mean" )
                             ),
                             Frequencies( 0 ),
                         )
                     )
                 )
             )
         )
     );
```

Examples of Combined Display Boxes

The following example generates a sample of many controls illustrated in the previous
sections.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Window Controls",
    Line Up Box( N Col( 2 ), Spacing( 3 ),
        Panel Box( "Panel Box",
            Check Box( {"Check Box 1", "Check Box 2"} ),
            Radio Box( {"Radio Box 1", "Radio Box 2"} ),
            Combo Box( {"Combo Box"} ),
            Button Box( "Button Box" ),
            List Box( {"List Box 1", "List Box 2"} )
        ),
        Col List Box( "all" )
```

```
    )
  );
```

Figure 11.16 Example of Many Interactive Display Elements

Example of Slider Boxes and Range Slider Boxes

The following example combines a graph with two sliders and a button by gluing the graph box, two horizontal boxes, and a button together in a vertical list box:

```
// Slider LogNormal
lU = 1;
lS = 2;
win = New Window( "LogNormal Density",
  V List Box(
    gr = Graph Box( // define the arguments and script for the graph box
      Frame Size( 500, 300 ),
      X Scale( 0.01, 3 ),
      Y Scale( 0, 4 ),
      XAxis( Show Major Grid ),
      YAxis( Show Major Grid ),
      Y Function(
        Exp( -(Log( x ) - Log( lU )) ^ 2 / (2 * lS ^ 2) ) / (lS * x *
        Sqrt( 2 * Pi() )),
          x
      );
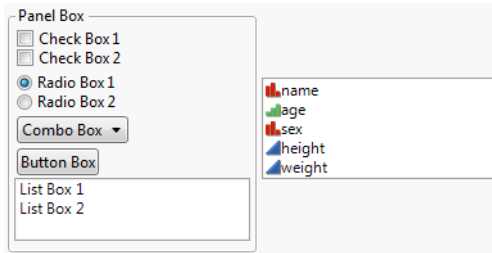
      /* display the specified text at 1 on the x axis
      and .5 on the y axis */
      Text( {1, .5}, "u= ", lU, " s= ", lS );
            ),

    /* define the minimum and maximum range
    insert the value of lU and updates the display as the slider moves */
    H List Box( Slider Box( 0, 4, lU, gr << Reshow ), Text Box( "mu" ) ),
      H List Box(
      Slider Box( 0, 4, lS, gr << Reshow ),
      Text Box( "sigma" )
```

```
        ),
        Button Box( "Reshow", gr << Reshow )
    )
);
Show( gr );
gr << Reshow; // update the display of the Graph Box
```

Figure 11.17 Example of Sliders and Buttons in a Report Window

The following script uses a `Range Slider Box()` instead of a `Slider Box()` to perform the same function:

```
lLow = 1;
lHigh = 2;
win = New Window( "Range Slider",
    V List Box(
        gr = Graph Box(
            Frame Size( 500, 300 ),
            X Scale( 0.01, 4 ),
            Y Scale( 0, 4 ),
            XAxis( "Show Major Grid" ),
            YAxis( "Show Major Grid" ),
            X Function( lLow, x );
            X Function( lHigh, x );
            Text( {1, .5}, "Low= ", lLow, " High= ", lHigh );
        ),
        V List Box(
            Text Box( "Low and High" ),
            sb = Range Slider Box( 0, 4, lLow, lHigh, gr << Reshow )
```

```
                )
            ),
            Button Box( "Reshow",
                lLow = 1;
                lHigh = 2;
                gr << Reshow;
                sb << Reshow;
            )
        );
        Show( gr );
        gr << Reshow;
```

Figure 11.18 Example of Using Range Slider Box

The script below is similar but uses `Global Box()` as an editable text box instead of two
`Slider Box()` controls:

```
        // Global LogNormal
        lU = 1;
        lS = 2;
        win = New Window( "LogNormal Density",
            V List Box(
                gr = Graph Box( // define the arguments and script for the Graph Box
                    Frame Size( 500, 300 ),
                    X Scale( 0.01, 3 ),
                    Y Scale( 0, 4 ),
                    XAxis( "Show Major Grid" ),
                    YAxis( "Show Major Grid" ),
                    Y Function(
                        Exp( -(Log( x ) - Log( lU )) ^ 2 / (2 * lS ^ 2) ) / (lS * x *
```

```
                    Sqrt( 2 * Pi() )),
                x
            );
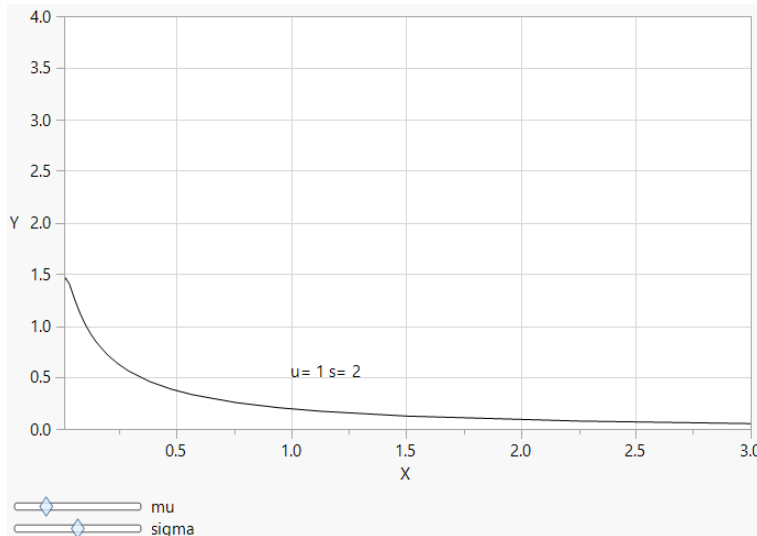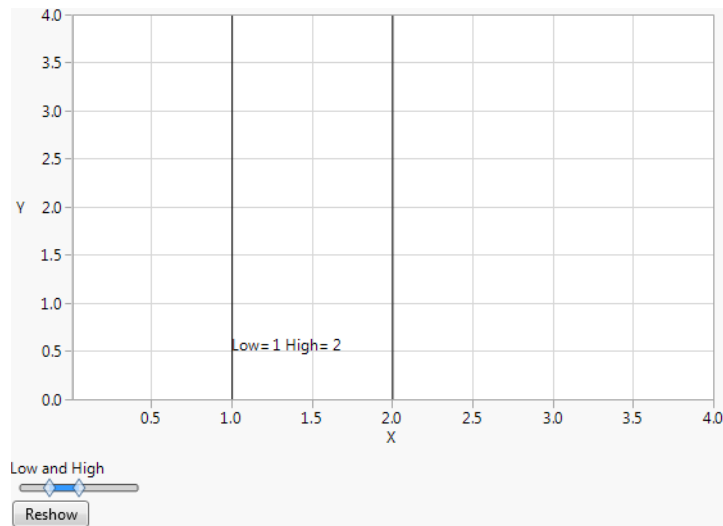
            // display the specified text at 1 on the x and y axes
            Text( {1, 1}, "u= ", lU, " s= ", lS, " or type new values below" );
        ),

        // display the lU variable value in the first H List Box
        H List Box( Global Box( lU ) ),

        // display the lS variable value in the second H List Box
        H List Box( Global Box( lS ) )
    )
);
```

Figure 11.19 Example of Using a Global Box Instead of Sliders

IU = 1
IS = 2

Example of a Calendar and Date Selector

You can create a calendar or date selector using `Calendar Box()` and `Number Edit Box()`.

Calendar Box

With `Calendar Box()`, you can select dates from a calendar and input the year. The following example creates a calendar with October 5, 1989 initially selected. You can select any date 60 days before or 60 days after October 5, 1989.

```
win = New Window( "Calendar Box", cal = Calendar Box() );
date = Date MDY( 01, 04, 2021 );
```

```
cal << Date( date );
cal << Show Time( 0 ); // omit the time
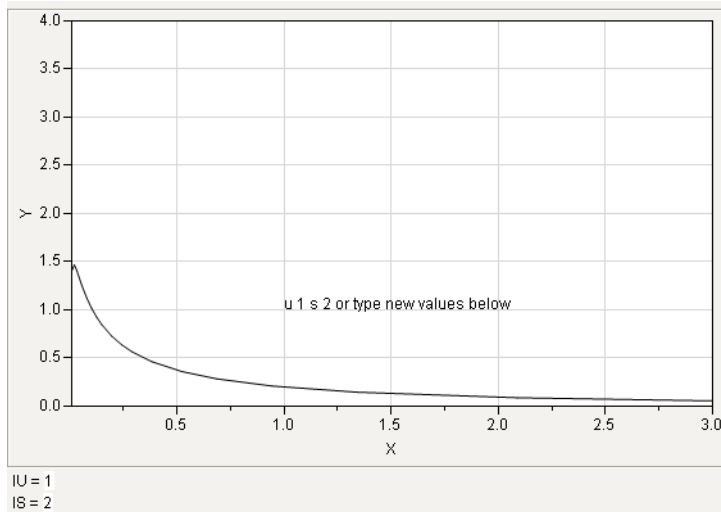
/* earliest date that can be selected is 60 days before 01/04/2021
"start" truncates the value so the time is not included */
cal << Min Date( Date Increment(date, "Day", -60, "start" ) );

// latest date that can be selected is 60 days after 01/04/2021
cal << Max Date( Date Increment(date, "Day",  60, "start" ) );
// print the abbreviated date to the log
cal << Set Function( Function( {this}, Print( Abbrev Date(this << Get Date())
  ) ) );
```

Figure 11.20 Creating a Calendar

Number Edit Box

With Number Edit Box(), you can enter a date in the box or click the calendar button to select the date from a calendar.

The following example shows how to create an interactive calendar using Number Edit Box(). The user enters a date in the box or clicks the calendar to select a date from the calendar. The number appears in a text box below the number edit box. "01/04/2021" is initially selected.

```
f = Function( {this}, // callback
  textbox << Set Text( Format( this << Get, "m/d/y" ) ) // get the date
);
win = New Window( "Date Selector",
  numbox = Number Edit Box( 0, 20, << Set Function( f ) ),
  textbox = Text Box( "" ) // specified date
);
numbox << Set Format(
  Format( "m/d/y", 12 ) // date format and number of characters displayed
  );
numbox << Set( Date MDY( 01, 04, 2021 ) ); // default date
```

Figure 11.21 Creating a Date Selector

Note that a callback is assigned to the f variable. In the callback, this refers to the number edit box, the box that calls the function later. Referring to this prevents the need for using the numbox variable within the callback. The Get message is sent to the this argument to get the specified date.

Example of Constructing a Summary Results Report

The following example uses the Summarize() function to collect summary statistics on the Height column of Big Class.jmp and then uses display box constructors to show the results in a nicely formatted window.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
    a = by( age ),
    c = count,
    sumHt = Sum( Height ),
    meanHt = Mean( Height ),
    minHt = Min( Height ),
    maxHt = Max( Height )
);
win = New Window( "Summary Results",
    Table Box(
        String Col Box( "Age", a ),
        Number Col Box( "Count", c ),
        Number Col Box( "Sum", sumHt ),
        Number Col Box( "Mean", meanHt ),
        Number Col Box( "Min", minHt ),
        Number Col Box( "Max", maxHt )
    )
);
```

This produces the summary results shown in Figure 11.22.

Figure 11.22 Producing a Customized Summary Report

Age	Count	Sum	Mean	Min	Max
12	8	465	58.125	51	66
13	7	422	60.2857	56	65
14	12	770	64.1667	61	69
15	7	452	64.5714	62	67
16	3	193	64.3333	60	68
17	3	200	66.6667	62	70

The following script saves the report as a Microsoft PowerPoint presentation.

```
Show Properties( win );
win << Save Presentation( "$DOCUMENTS/Summary.pptx" );
```

Interact with a Closing Window

A user can close a window by clicking the OK or Cancel button or by clicking the red Close button in the top corner of the window. A message can also be sent to close the window. On Validate and On Close expressions determine what happens you interact with the window.

- On Validate evaluates its expression *when the OK button is clicked*. The expression is not evaluated for any other button click or action. If the expression evaluates as true, the window closes. If the expression evaluates as false, the window is not closed. The script for an OK button runs when you click the button, no matter what the On Validate expression does.

- On Close evaluates its expression *right before the window closes*. The On Close return value is like the On Validate return value and can prevent the window from closing. However, the best practice for On Close is to always return 1 so that you do not accidentally create a window that cannot be closed.

  If the window closes, On Close is evaluated. On Close is not evaluated if the On Validate expression returns false, because the window is not closed.

Tips:

- On Close works for modal or non-modal windows. On Validate works only for modal windows.

- Do not use On Close for validation because it does not distinguish between OK and Cancel. Use On Close to clean up resources (such as invisible data tables) because it always runs when the window closes.

- Do not use On Validate to clean up resources. Use On Close instead or in addition to On Validate, because On Validate does not run for the Cancel button or red Close button.

On Close() Example

Suppose that a non-modal window is a report window, and the resource that needs cleaning up is an invisible data table. The On Close script closes the data table. This is good because the user might not know that the data table is open and has no easy way to close it.

```
dt = New Table( "Untitled", << Invisible,
    New Column( "x", Set Values( [1, 2, 3, 4] ) ),
    New Column( "y", Set Values( [4, 2, 3, 1] ) )
);

dt << Bivariate(
    y( dt:y ), x( dt:x ), <<On Close( Close( dt, "nosave" ) )
```

```
);
```

Before the script is run, no data tables are open. The Untitled invisible table is created and displayed in the Home Window and in the Data Tables list above the script. Then the graph is created. After you close the report window, the invisible table is closed by `On Close`.

On Validate() Example

Suppose that a modal window contains a question that the user must answer before closing the window. The following example shows how to validate the input when the user closes the window.

```
win = New Window( "Validate Example",
   <<Modal,
   <<Return Result,
   <<On Validate(
      If( (!Is Missing( variablebox << Get ) & 40 <= variablebox << Get <= 50),
         ( 1 ), // else the user did not enter the correct answer
         tb << Set Text(
            Match( Random Integer( 1, 3 ),
               1, "Are you sure?",
               2, "Guess again.",
               3, "Please read the question."
            )
         );
         ( 0 );
      )
   ),
   Text Box( "I'm thinking of a number between 40 and 50. What's your guess?"
   ),
   variablebox = Number Edit Box( . ),
   H List Box( Button Box( "OK" ), Button Box( "Cancel" ) ),
   tb = Text Box( "" )
);

New Window( "Modal Window",
   <<Modal,
   Text Box(
      If(

         // user clicked Cancel or the red Close button
         win["Button"] == -1, "Try again when you feel better.",
         win["variablebox"] == 42, "Good job.", // user entered 42
         "Take the test again." // user did not enter 42 and clicked OK
      )
   )
);
```

Update an Existing Display

Sometimes, you do not know how many display boxes will appear in a future report. For example, you might write a generic script that analyzes and reports on one or more variables. You do not know how many display boxes are needed, because the number of variables can change from one run of the script to the next.

The following sections describe how to add and delete display boxes from a report using `Append()`, `Prepend()`, `Delete()`, and `Sib Append()`.

Append

Use the `Append` message to add a display box to the bottom of an existing display. In the script, construct a single, empty box, then append boxes to it for each variable in the analysis. You can also use the `Append` message to modify an existing display with custom information or organization.

The following code example assumes that there is a list of effect names in the variable `effectsList`, and that each one corresponds to a column in a matrix `varprop`. In other words, `effectsList[1]` is the label for `varprop[0,1]`; `effectsList[2]` is the label for `varprop[0,2]`; and so on.

```
varprop = [0 1 2, 3 4 5, 6 7 8];
effectsList = {"one", "two", "three"};
```

First, an empty `Outline Box` containing an `H List Box` is created. The interior empty container is given the name `hb`:

```
win = New Window( "H List Box Example",
    Outline Box( "Variance Proportions", hb = H List Box() )
);
```

Then, a `for` loop steps through the `effectsList` and adds a `Number Col Box` for each element of `effectsList`:

```
For( i = 1, i <= N Items( effectsList ), i++,
    Eval(
        Substitute(
            Expr(
                hb << Append(
                    Number Col Box( effectslist[i], varprop[0, i] )
                )
            ),
            Expr( i ), i
        )
    )
);
```

Sib Append() is similar to Append() but makes the display box a sibling of the last child display box. For an example that compares Append() to Sib Append(), see Figure 11.24.

Prepend

The Prepend message works just like Append but adds the item at the beginning of the display box rather than at the end. If the display box is one of several that do not allow appending, then Prepend delegates the command to a child display box that can accept the command. It is fine to apply it to the top of the tree.

For example, the following example creates a Bivariate report with a button box at the top.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( height ), X( weight ), Fit Line );

/* select the first outline box in the biv report layer
send the Prepend message to the first outline box */
(biv << Report)(Outline Box( 1 )) << Prepend(

    // prepend a button box that fits a quadratic curve
    Button Box( "Click here for curve", biv << Fit Polynomial( 2 ) )
);
```

Click the **Click here for curve** button to add a quadratic curve to the graph.

You can accomplish the same thing by appending the button to the top of the tree:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( height ), X( weight ), Fit Line );
biv << Report << Prepend( // send the prepend message to the report

  // prepend the button box to the top of the report
  Button Box( "Click here for curve", biv << Fit Polynomial( 2 ) )
);
```

Delete

The Delete message removes the specified display box and all its children from the report. This is useful with the Append and Prepend messages for building completely dynamic displays. In the example below, a text box is replaced with another text box. In this case, the script could have used Set Text, but many display boxes cannot change their content.

```
win = New Window( "X",
  list = V List Box(
      t1 = Text Box( "t1" ),
      t2 = Text Box( "t2" )
  )
);
```

```
t1 << Delete;
list << Append( t1 = Text Box( "t1new" ) );
```

Note: Avoid deleting display boxes from a JMP platform. Send the Hide(1) message or the
Visibility("Hidden") message to the display box that you want to hide.

Sib Append

You can use the Sib Append message to add a display box immediately after an existing
display box. Figure 11.23 shows two picture box trees. Picture Box(1) holds the Bivariate
scatterplot. Picture Box(2) holds the Fit Mean menu, determined by seeing the green line
and the Fit Mean text box.

Suppose you wanted to insert a text box in between these two boxes. You want to append a
sibling to Picture Box(1), so you send it the Sib Append message:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Mean( {Line Color( {57, 177, 67} )} )
);

/* the report function takes the biv object and returns a report object.
 append the text box immediately after the first picture box */
Report( biv )[Picture Box( 1 )] << Sib Append( Text Box( "Hello There" ) );
```

Figure 11.23 Appending a Sibling Text Box

Note: In Figure 11.23, the appended sibling text box appears below the first picture box in the Show Tree Structure View window. In the classic Show Tree Structure View window, siblings appeared next to each other in the display tree. See "View the Display Tree" on page 521.

Figure 11.24 shows the difference between and `Append()` and `Sib Append()`. `Append()` makes the display box a *child* of the parent display box. `Sib Append()` makes the display box a *sibling* of the last display box.

Figure 11.24 Append (Left) and Sib Append (Right)

Note: If the parent display box is not a V List Box(), Sib Append() wraps that display box in V List Box().

Set Function and Set Script

You can use the Set Script message to have a display box control (for example, a button box or combo box) run a script when it is clicked with the mouse.

```
win = New Window( "Set Script Example",
   ex = Button Box( "Press me" )
);
ex << Set Script( Print( "Pressed" ) );
```

The preceding script prints the following text to the log when the button box is clicked:

```
"Pressed"
```

Alternatively, you can use the Set Function message to have a display box control run a specific function where the first argument is the specific display box. Set Function enables you to build more object-oriented scripts and to create a larger system.

```
win = New Window( "Set Function Example",
   Button Box( "Press me",

      // functions specified with Set Function get 'this' display box
      << Set Function(
         Function(
            {this},
            this << Set Button Name( "Thanks" )
         )
      )
   )
);
```

The preceding script creates a button box with the name "Press me". When the button is clicked, the function called by Set Function changes the name to "Thanks".

If you have multiple buttons that can be serviced by the same script, Set Function is much simpler than Set Script because it tells you which button was pressed. In this example, both check boxes use the script that begins on the first line.

```
f = Function( {this, idx}, // idx just changed
	Write(

	/* <<Get Items returns a list of all items; indexing that list with idx
	is the name of the item that is changing */
		"\!n changing item=" || (this << Get Items())[idx] || " to " || Char(

	// <<Get returns the new value of that item
	this << Get( idx ) ) ||
		"\!n new selection=" || Char( this << Get Selected ) || "\!n"
	)// <<Get Selected returns a list of the currently checked (selected) items
);
New Window( "Example",
	H List Box( // 'f' is the 'named' function, used twice
		V List Box( Text Box( "Column 1" ), Check Box( {"a", "b"}, <<Set
	Function( f ) ) ),
		Spacer Box( size( 50, 50 ) ),
		V List Box( Text Box( "Column 2" ), Check Box( {"c", "d"}, <<Set
	Function( f ) ) )
	)
);
```

In the following example of Set Function, the same function is used to create the buttons.

```
New Window( "Cash Box",
	V List Box(
		H Center Box( TB = Text Box() ), // TB will show total of coins
		LB = Lineup Box( N Col( 3 ) ), // LB will hold buttons, added below
		H Center Box( // CLEAR button will reset the total
			Button Box( "CLEAR",
				total = 0;
				TB << Set Text( Char( total ) );
			)
		)
	)
);
coins = {1, 5, 10, 25, 50, 100}; // coins is a list of the button labels
total = 0;
```

```
/* loop creates the buttons and shows that the same function
is used for each */
For( iButton = 1, iButton <= N Items( coins ), iButton++,
  LB << Append( Button Box( Char( coins[iButton] ), Set Function(
  buttonFunction ) ) )
);
buttonFunction = Function( {this},
  total = total + Num( this << Get Button Name );
  TB << Set Text( Char( total ) );
);
```

The function called for any button, except CLEAR, uses the button's name to determine what to do. Rather than using the button's name, you might want to use a sibling display box by this<<sib or perhaps (this<<parent)<<child to find the first sibling of this button.

Notes:

- The Set Script and Set Function messages work for button boxes, calendar boxes, check boxes, combo boxes, list boxes, popup boxes, radio boxes, range slider boxes, slider boxes, and spin boxes.

- You cannot use both Set Script and Set Function at the same time. Use Set Function if you need to reference a specific display box object.

Get and Set Selected Values of Display Elements That Return Lists

You can use the Set Selected(Item Number, <State>, <Run Script(0|1)>) message to pre-select an item. You can use this message separately to a saved display box reference, or you can specify it inline as a list box << message.

To retrieve the selected value, use the Get Selected message, which returns the value of the selected item. The Get Selected Indices message returns the index number of the selected item.

```
antennaList = {"Dish","Helical","Polarizing","Radiant Array"};

win = New Window( "Test List", // method 1: display box reference
  listObj = List Box(
    antennaList,
    Print(
      "iList",
      listObj << Get Selected,
      listObj << Get Selected Indices
    )
  )
);
listObj << Set Selected( 2, 1 );
```

```
win = New Window( "Test List", // method 2: inline
  listObj = List Box(
     antennaList,
     <<Set Selected( 2, 1 ),
     Print(
        "iList",
        listObj << Get Selected,
        listObj << Get Selected Indices
     )
  )
);
```

Both of these scripts print the following text to the log:

```
"iList"
{"Helical"}
{2}
```

In the preceding examples, the Print expression is executed when the Set Selected message is completed. To prevent the script from running, include Run Script(0) as the last argument. Run Script(0|1) controls whether a display box on-change script runs after a Set or Set Selected message.

```
antennaList = {"Dish","Helical","Polarizing","Radiant Array"};
win = New Window( "Test List",
  listObj = List Box(
     antennaList,
     Print(
        "iList",
        listObj << Get Selected,
        listObj << Get Selected Indices
     )
  )
);
listObj << Set Selected( 2, 1, Run Script( 0 ) );
```

With Run Script(1), the script is executed when the Set message is completed, even if the value is unchanged. (The script does not run again if the user selects the same value.) With Run Script(0), the script does not run.

On most interactive display boxes, the script does not run on most interactive display boxes if you leave out Run Script(). However, on List Box(), the script runs by default, which is consistent with previous behavior.

Send Messages to Constructed Displays

When you assign a constructed display to a name, that name becomes a reference to the window, which in turn owns the display boxes inside it. Using subscripts, you can then send messages to the display boxes inside the window.

For example, the following script creates an interactive sine wave graph. The script automates the interaction by sending messages to the frame box inside the window. Note the frame box assignment to *tf* in the middle of the script).

```
amplitude = 1;
freq = 1;
phase = 0;
win = New Window( "Wiggle Wave",
  Graph Box(
    Frame Size( 500, 300 ),
    X Scale( -5, 5 ),
    Y Scale( -5, 5 ),
    Y Function( amplitude * Sine( x / freq + phase ), x );
    Handle(

      /* current values of phase and amplitude position
      the square handle.
      script begins after the last comma after amplitude */
      phase,
      amplitude,

      /* x and y have been set to the handle's position
      but the handle doesn't move unless the values specified
      in the first two arguments are updated */
      phase = x;
      amplitude = y;
    );

    // Handle works similarly except that the y value is .5
    Handle( freq, .5, freq = x );

    Text( // display a text string in the graph
      {3, 4},
      "amplitude: ",
      Round( amplitude, 4 ), // display the current value
      {3, 3.5},
      "frequency: ",
      Round( freq, 4 ), // display the current value
      {3, 3},
      "phase: ",
      Round( phase, 4 ) // display the current value
```

```
        );
      )
    );
    tf = win[Frame Box( 1 )]; // get the frame box (the graph)
    For( amplitude = -4, amplitude < 4, amplitude += .1,// animate the amplitude
      tf << Reshow // force the graph to update
    );
    amplitude = 1; // use for loops for more complex movement
    freq = 1;
    phase = 0;
    For( i = 0, i < 1000, i++,
      amplitude += (Random Uniform() - .5);
      amplitude = If(
        amplitude > 4, 4,
        amplitude < -4, -4,
        amplitude
      );
      freq += (Random Uniform() - .5) / 20;
      phase += (Random Uniform() - .5) / 10;
      tf << Reshow;
      Wait( .05 );
    );
```

Construct Display Boxes That Contain Platforms

Another type of display that you might want to construct is your own combination of results
from the analysis platforms in JMP. Write the platform script inside a display box and
assemble the display boxes into a window. For ease in routing messages to it later, assign the
whole script to a reference.

The following example creates several graphs and reports in one window:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
csp = New Window( "Platform Example",
  Outline Box( "Analyses of height in Big Class.jmp",
    H List Box(
      cc = Control Chart(
        Chart Col( Height, "Individual
        Measurement", "Moving Range" ),
        K Sigma( 3 )
      ),
      dist = Distribution( Columns( Height ) )
    )
  )
);
```

Figure 11.25 Example: Multiple Graphs in One Report Window

Now you can work with the window by sending messages to the reference *csp*. This is a display box reference, whose capabilities are similar to those of a Report for a platform. You can use multiple-argument subscripting to locate specific items within the outline tree:

```
csp["Control ?", "moving range ?"] << Close;
csp["Dist?", "quantiles"] << Close;
```

Notice that the preceding script not only assigned the whole window to a reference (csp) but also assigned the platform launch scripts to names (cc and dist) within their display boxes. This makes it easy to route messages to the platforms. You could, in turn, get the reports for these and have yet another way to manipulate display boxes. The following are equivalent messages that reopen the nodes:

```
rcc = cc << Report;
rdist = dist << Report;
rcc["moving range ?"] << Close;
rdist["quantiles"] << Close;
```

You can send messages directly to the platform references themselves. Find out your options by searching for "Distribution" and "Control Chart" in the JMP Scripting Index. For example, Control Chart provides a message called Needle that creates a needle chart. Distribution provides a message called Normal Quantile Plot.

To execute these options from JSL, send them as messages to the platform references:

```
cc << Needle;
dist << Normal Quantile Plot;
```

Figure 11.26 Changing a Custom Report

Counting the Number of Child Boxes

Suppose that you want to count the number of child text boxes in a list box. The most reliable way is to use XPath to find a specific string in the text boxes and then use N Items() to count the number of text boxes.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate(
```

```
      Y( :weight ),
      X( :height ),
      Group By( :age ),
      Fit Line( {Line Width( 3 )} )
   );
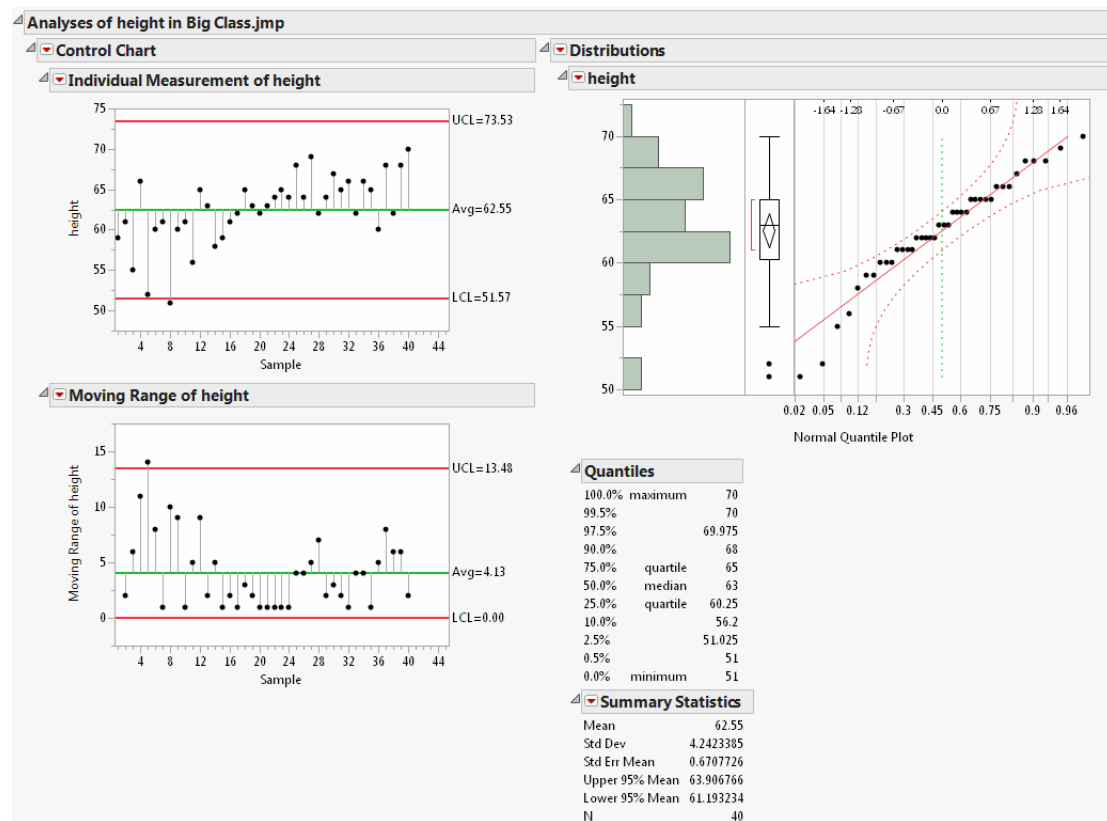
   /* start with the root list box, find a text edit box
   that starts with "Linear Fit age==", and count the number of children */
   N Items( biv << XPath( "//ListBox[starts-with(TextEditBox,'Linear Fit
      age==')]" ) );
```

Examples of Creating a Dashboard from Two Reports

In JMP, you create dashboards in Dashboard Builder. See *Using JMP*. This section shows how to write scripts to create dashboards.

Dashboard with One Row of Reports

In the report window, reports appear in tab page boxes. You reposition reports in a dashboard by clicking the tab title and dragging the tab page to a highlighted drop zone. The highlighted drop zones indicate how to arrange reports horizontally, vertically, or in tabs. If a valid drop zone is not selected, the box returns to its original location.

This example shows how to write a script that creates a dashboard with two reports in a row:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "My Dashboard",
   H Splitter Box( // place reports in a splitter box
   Size( 700, 400 ),
      Tab Page Box( // place report in a tab page box
         Title( "Bivariate Fit of weight by height" ),
         dt << Bivariate( // Bivariate report
            Y( :weight ),
            X( :height ),
            Fit Line,
            Report View( "Summary" ) // show only the graph
         ),
         <<Moveable( 1 ) // make the report moveable
      ),
      Tab Page Box(
         Title( "Oneway Analysis of height by sex" ),
         dt << Oneway( // Oneway report
            Y( :height ),
            X( :sex ),
            Means( 1 ),
            Mean Diamonds( 1 ),
```

```
            Report View( "Summary" )
        ),
        <<Moveable( 1 )
    ),
    /* each tab box can be dragged to a different location
    required for moveable display boxes */
    <<Dockable( 1 )
  )
);
```

Figure 11.27 Dashboard with Two Reports

Note: For dockable and moveable tabs that contain a single platform, the top-level outline box is subsumed into the tab title. This means that the red triangle options now appear in the title bar of the tabs.

Dashboard with Tabs

The following script creates a dashboard that places each report on a tab:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "My Dashboard",
  Tab Box( // place reports in a tab box
    Tab Page Box( // place report in a tab page box
      Title( "Bivariate Fit of weight by height" ),
```

```
        dt << Bivariate( // Bivariate report
            Y( :weight ),
            X( :height ),
            Fit Line,
            Report View( "Summary" ) // show only the graph
        ),
        <<Moveable( 1 ), // make the report moveable
        <<Closeable( 1 ) // make the report closeable
    ),
    Tab Page Box(
        Title( "Oneway Analysis of height by sex" ),
        dt << Oneway( // Oneway report
            Y( :height ),
            X( :sex ),
            Means( 1 ),
            Mean Diamonds( 1 ),
            Report View( "Summary" )
        ),
        <<Moveable( 1 ),
        <<Closeable( 1 )
    ),
    /* each tab box can be dragged to a different location
    required for moveable display boxes */
    <<Dockable( 1 ),
    <<Set Overflow Enabled( 0 ) // both tab pages appear
  )
);
```

Figure 11.28 Dashboard with Tabs

In a dashboard with several tabs, the window widens to show all tabbed reports. You might want to show only the first report and then let the user select the other tabbed reports from a list. Change `Set Overflow Enabled( 0 )` near the end of the script to `Set Overflow Enabled( 1 )`. Figure 11.29 shows the result.

Figure 11.29 Dashboard with Overflow Tabs

How to Dock Tabs in Tab Boxes

A docking layout is a collection of `Tab Page Box()` objects and docking containers that enable reports to be rearranged. In the report window, you drag a docked tab to a highlighted drop zone to create a new layout. Figure 11.27 shows two docked reports. In Figure 11.30, the Oneway report was dragged to the highlighted drop zone to display the reports in one column.

Figure 11.30 Docking a Tab in a Dashboard

You combine `Tab Box()` with `Splitter Box()` to create docked tabs. Tab boxes and splitter boxes must directly contain other docking containers or the tab pages that define the content. Note the stacked boxes from a portion of the preceding example:

```
H Splitter Box( // place reports in a splitter box
    Size( 700, 400 ),
    Tab Page Box( // place report in a tab page box
```

The following expressions specify that the tab box and splitter box are docking containers, within which tab pages can be docked:

```
Tab Box << Set Dockable();
Splitter Box << Set Dockable();
```

In the report window, you reposition tab pages by clicking the tab title and dragging the tab page to a highlighted drop zone. The highlighted drop zones indicate how to arrange reports horizontally, vertically, or in tabs. If a valid drop zone is not selected, the box returns to its original location.

Closing Tabs

The following expression creates a close button ✖ to the right of the report title on a closeable tab page:

```
Tab Page Box << Set Closeable();
```

Clicking the close button ✖ deletes the tab and its contents. Parent containers might also be removed if they are no longer necessary (for example, if there are no more tabs in a tab box or only a single child in a splitter box).

Use the `Tab Page Box <<Set Close` message to run a JSL script when the close button ✖ is clicked. A return value of "1" indicates that the tab is closed. A return value of "0" indicates that closing the tab is not allowed at that time. Closing the tab might not be allowed as the result of a user prompt or based on the state of the program.

Controlling Wide Tabs

When a docking layout includes tab page boxes, you may place many pages in a single `Tab Box()` container. In this case, the titles may become wider than the actual content or wider than desired in the overall display. The `Tab Box <<Set Overflow Enabled( 0|1 )` message specifies that the tab titles should not cause the tab box to grow wider. If the titles do not all fit, a pop-up button appears in the upper right corner of the tab box to select tabs that are in the "overflow list". See Figure 11.29 on page 599 for an example.

Example of Creating a Cluster Platform Launch Window

The following example creates a simplified replica of the Cluster platform launch window, which then launches the platform with the given arguments.

Note: Some functions (`Recall` and `Help`) are not implemented in the script, so an alert window is shown when the Recall and Help button are clicked. In addition, switching from Hierarchical to K-Means clustering does not change anything, unlike the real Cluster launch window.

```
dt = Open( "$SAMPLE_DATA/Birth Death.jmp" );
nc = N Col( dt ); // number of columns in the window
lbWidth = 130; // width of the window

// define the method list
methodList = {"Average", "Centroid", "Ward", "Single", "Complete"};
notImplemented = Expr(
  win = New Window( "Feature Not Implemented Yet", <<Modal, Button Box( "OK"
  ) )
);
clusterDlg = New Window( "Clustering", // create the window
  <<Modal,
  Border Box( Left( 3 ), Top( 2 ),
    V List Box(
      Text Box( "Finding points that are close, have similar values" ),
      H List Box(
        V List Box(
          Panel Box( "Select Columns",
            colListData = Col List Box(
              All,
              width( lbWidth ),
              NLines( Min( nc, 10 ) )
            )
          ),
          Panel Box( "Options",
            V List Box(
              comboObj = Combo Box(
                {"Hierarchical", "K-Means"},
                <<Set( 1 )
              ),
              Panel Box( "Method",
                methodObj = Radio Box( methodList, <<Set( 3 ) )
              ),
              checkObj = Check Box( {"Standardize Data"}, <<Set( 1, 1 ) )
            )
          )
        ),
        Panel Box( "Cast Selected Columns into Roles",
          Line Up Box( N Col( 2 ), Spacing( 3 ),
            Button Box( "Y, Columns",
              colListY << Append( colListData << GetSelected )
            ),
            colListY = Col List Box(
              width( lbWidth ),
              NLines( 5 ),
              "numeric"
```

```
      ),
      Button Box( "Ordering",
         colListO << Append( colListData << GetSelected )
      ),
      colListO = Col List Box(
         width( lbWidth ),
         NLines( 1 ),
         "numeric"
      ),
      Button Box( "Label",
         colListL << Append( colListData << GetSelected )
      ),
      colListL = Col List Box( width( lbWidth ), NLines( 1 ) ),
      Button Box( "By",
         colListB << Append( colListData << GetSelected )
      ),
      colListB = Col List Box( width( lbWidth ), NLines( 1 ) )
   )
),
Panel Box( "Action",
   Line Up Box( N Col( 1 ),
      Button Box( "OK",
         If( (comboObj << Get) == 1,
            Hierarchical Cluster(
               Y( Eval( colListY << GetItems ) ),
               Order( Eval( colListO << GetItems ) ),
               Label( Eval( colListL << GetItems ) ),
               By( Eval( colListB << GetItems ) ),
               Method( methodList[methodObj << Get]
               ),
               Standardize( checkObj << Get( 1 ) )
            ),
            KMeansCluster( Y( colListY << GetItems ) )
         );
         clusterDlg << Close Window;
      ),
      Button Box( "Cancel", clusterDlg << Close Window ),
      Text Box( " " ),
      Button Box( "Remove",
         colListY << RemoveSelected;
         colListO << RemoveSelected;
         colListL << RemoveSelected;
         colListB << RemoveSelected;
      ),
      Button Box( "Recall", notImplemented ),
      Button Box( "Help", notImplemented )
```

```
                    )
                )
            )
        )
    )
);
```

Figure 11.31 The Cluster Launch Window

Example of Constructing a Custom Platform

An example in "Manipulating Expressions" on page 268 in the "Programming Methods" chapter, showed how to use the Substitute Into() function to input coefficients for a quadratic polynomial into the quadratic formula and then use the formula to calculate the roots of the polynomial. That example required specifying the coefficients as arguments to Substitute Into().

The section "Construct a Column Dialog" on page 611 shows an example to collect coefficients from the user using a modal dialog box.

This section further develops the example into a complete customized platform that first displays a dialog box to ask for coefficients, finds the roots, and then displays the results along with a graph in a custom window.

```
// create a window to collect coefficients from the user
myCoeffs = New Window( "Find the roots for the equation",
    <<Modal,
    H List Box(
        a = Number Edit Box( 1 ),
        Text Box( "*x^2 + " ),
        b = Number Edit Box( 2 ),
        Text Box( "*x + " ),
        c = Number Edit Box( 1 ),
        Text Box( " = 0" )
```

```
   ),
   Button Box( "OK",
      a = a << Get;
      b = b << Get;
      c = c << Get;
      Show( a, b, c );
   ),
   Button Box( "Cancel" )
);
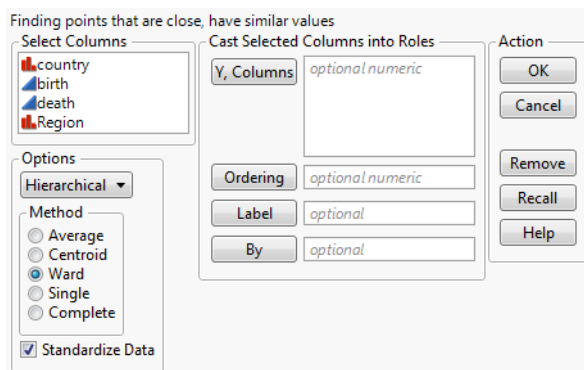
/* calculate the results: The quadratic formula is
x=(-b + - sqrt(b^2 - 4ac))/2a. Plug the coefficients into
the quadratic formula */
x = {Expr(
   (-b + Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
), Expr(
   (-b - Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
)};
xx = Eval Expr( x );


results = Expr( // store the solution list
/* test whether real roots were found and make an appropriate display
if yes (for example, with the window's defaults), show roots and a graph */
   xmin = xx[1] - 5;
   xmax = xx[2] + 5;
   ymin = -20;
   ymax = 20;
   win = New Window( "The roots of a quadratic function",
      V List Box(
         Text Box( "The real roots for the equation " ),
         Text Box( "     " || Expr( po ) || " = 0" ),
         H List Box( Text Box( "are x=" ), Text Box( xxx ) ),
         Text Box( " " ), // to get a blank line
         Graph Box(
            Frame Size( 200, 200 ),
            X Scale( xmin, xmax ),
            Y Scale( ymin, ymax ),
            Line Style( 2 ),
            H Line( 0 ),
            Line Style( 0 ),
            Y Function( polynomial, x ),
            Line Style( 3 ),
            Pen Color( 3 ),
            V Line( xx[1] ),
            V Line( xx[2] ),
```

```
                Marker Size( 2 ),
                Marker( 0, {xx[1], 0}, {xx[2], 0} )
            )
        )
    );
);
error = Expr(

    /* if no (for example, with a=3, b=4, c=5), put up an error
    window with a helpful graph */
      win = New Window( "Error",
        V List Box(
            Text Box( " " ),
            Text Box( "  Polynomial " || po || " has no real roots.  " ),
            Text Box( " " ),
            Text Box( "Examine a graph of the function to get an idea why." ),
            Graph Box(
                Frame Size( 200, 200 ),
                X Scale( -20, 20 ),
                Y Scale( -20, 20 ),
                Line Style( 2 ),
                H Line( 0 ),
                Line Style( 0 ),
                Y Function( polynomial, x )
            )
        )
      )
);

    /* either way, the script needs to have some strings ready
    rewrite the polynomial with the coefficients specified */
    polynomial = Expr( a * x ^ 2 + b * x + c );

    // store this instance of the polynomial as a string
    po = Char( Eval Expr( polynomial ) );

    // store the solution list as a string
    xxx = Char( Eval Expr( x ) );

    If( Is Missing( xx[1] ) | Is Missing( xx[2] ), // now it's ready for a test
        error,
        results
    );
```

When you run the preceding script, you first see a window like this:

Figure 11.32 A Custom Platform Launch Window

Clicking **OK**, displays a results window (Figure 11.33, left), with either the roots or an error message. Rerun the script, input 5, 4, and 5 respectively and click **OK**. Note that JMP displays an error message (Figure 11.33, right).

Figure 11.33 The Custom Platform's Report

Example of Specifying Text and Background Colors

The markup argument in a display box enables you to write HTML code to specify text and background colors. The color can be a hex value (starting with "0x" or "#") or a known JMP color. Here is an example:

```
New Window( "Colors",
  Text Box(
    "This is <font color=\!"Red\!">red text and this is <font
  color=\!"0x00ff00\!">green text and this is <font color=\!"#0000FF\!">blue
  text</font>.",
    <<markup
  ),
  Text Box(
    "This is <background color=\!"#ff0000\!"> red text and this
  is<background color=\!"Green\!"> green text and this is<background
  color=\!"0x0000FF\!"> blue text</background>",
    <<markup
  )
);
```

Figure 11.34 HTML Text and Background Colors

Modal Windows

As described in "Constructors for New Windows" on page 542, *modal* means that the user must interact with the window immediately. Clicking outside the window produces an error sound. Script execution stops until the user responds to the window.

JMP provides two types of modal window functions:

- New Window() with the Modal message. This creates a new window in which you arrange other display boxes.

- Column Dialog() constructs a window for column role assignments as found in many launch windows. The specialized constructors for Column Dialog() mean that you have less flexibility than with New Window(). However, Column Dialog() lets you easily create a launch window.

Construct a Modal Window

When you submit a script with a modal window, JMP draws the window, waits for the user to make choices and click **OK**, and then stores a list of the variables with their values. Any attempt to click outside the window produces an error sound, and script execution is suspended until the user clicks **OK** or **Cancel**.

The Column Dialog() function is specifically intended to prompt users to choose columns from the current (topmost) data table. Windows created by Column Dialog() are also modal windows.

Tips:

- Put all the modal windows near the beginning of the script, if possible. This way, all the user interaction can be accomplished at once, and then users can leave JMP to finish its work unattended.

- Make sure your modal windows give the user enough information. Do not just present a number field. Tell users how the number is used. If there are limits for valid responses, say so.

A very simple modal window might request a value for one variable:

```
win = New Window( "Set a Value",
    <<Modal,
```

```
    Text Box( "Set this value" ),
    variablebox = Number Edit Box( 42 ),
    Button Box( "OK" ),
    Button Box( "Cancel" )
);
```

Notice that the argument to Number Edit Box() is the default value, 42.

Figure 11.35 Sample Modal Display Box

If you click **OK**, the window closes and {Button(1)} is returned. To get the value entered in the box, use the Return Result message and a subscript into the window. Note that the deprecated Dialog() returns the list of variable assignments the same way.

```
win = New Window( "Set a Value",
    <<Modal,
    <<Return Result,
    Text Box( "Set this value" ),
    variablebox = Number Edit Box( 42 ),
    Button Box( "OK" ),
    Button Box( "Cancel" )
);
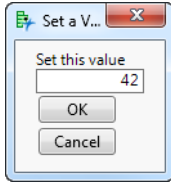
// create a subscript to the variablebox variable
Write( win["variablebox"] );
```

In this example, the user typed 33 in the number edit box. That value is stored in win.

If you click **Cancel**, the window closes, it returns {Button(-1)}, and the script continues.

To detect whether the user clicked Cancel, add the following expression:

```
If( win["Button"] == 1,
    Print( win["variablebox"] );
    ,
    Print("Canceled")
);
```

Note that a modal window must have at least one button. If you omit the Button Box() from your script, an OK button is automatically included. To omit the OK button, set the visibility to collapse.

```
win = New Window( "No OK Button",
```

```
<<Modal,
b = Button Box( "Close Window", b << Close Window ),
bb = Button Box( "OK", <<Visibility( "collapse" ) )
);
```

Note: Modal windows require at least one button for dismissing the window. You can use **OK**, **Yes**, **No**, or **Cancel** to label modal window buttons. If you do not include at least one of these buttons in a modal window, JMP adds an **OK** button.

Construct a Column Dialog

`Column Dialog()`, a variant of the deprecated `Dialog()` function, lets you prompt for column selections from the current data table or contextual data table. The `OK`, `Cancel`, and `Remove` buttons and the list of columns to choose from the data table are added automatically.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dlg = Column Dialog(
   Col ID = Col List( "X, Treatment", Max Col( 1 ) ),
   Group = Col List( "Group Factors" ),
   Split = Col List( "Y, Response" ),
   w = Col List( "Weight" ),
   H List( "Alpha", alpha = Edit Number( .05 ) )
);
```

Figure 11.36 Column Dialog

The following example returns a list similar to this one, depending on the user's choices:

```
{Col ID = {}, Group = {}, Split = {}, w = {}, alpha = 0.05, Button( -1 )}
```

For each destination list, a `Col List` clause must be a direct argument of `Column Dialog()` (not nested inside some other argument). An optional `MaxCol(`*n*`)` argument restricts the number of data columns that can be chosen to *n*. Lists are *always* returned, although they can sometimes be empty lists. You can include as many as twelve `Col List` clauses.

Other items permitted in the deprecated `Dialog()` are permitted in `Column Dialog()` and have the same functionality. You can specify the minimum and maximum number of columns that are allowed in a column dialog box with the `MinCol` and `MaxCol` arguments.

You can also specify the modeling type of the columns that are allowed to be selected (`Ordinal`, `Nominal` or `Continuous`). You can set the width of the list using `Select List Width(`*pixels*`)` argument. To set the width of the column list, use `Width(`*pixels*`)` inside the `Col List()` function.

The following example generates a column dialog box that only allows the selection of exactly one numeric column:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
rt_dlg = Column Dialog(
   cv = Col List( "Response To Test", MaxCol( 1 ), MinCol( 1 ), DataType(
   "Numeric" ) )
);
```

Figure 11.37 Restricting Selection of Columns

The `Data Type` choices are `Numeric`, `Character`, and `Any`.

In addition, use the `Columns` specification to pre-fill some column selections. For example, the following script assigns height to the X role and weight and age to the Y role:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dlg = Column Dialog(
   xCols = Col List( "X, Factors", Columns( :height ) ),
   yCols = Col List( "Y, Response", Columns( :weight, :age ) )
);
```

Comparison of Column Dialog and New Window

`New Window()` and `Column Dialog()` enable you to construct a launch window that has a column selector. Use `Column Dialog()` to construct a simple launch window. Several items are added automatically to column dialogs: the Remove and Cancel buttons, the list of columns, and an OK button (if no buttons are explicitly defined).

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dlg = Column Dialog(
  x = ColList( "X", Max Col( 1 ) ),
  y = ColList( "Y", Max Col( 1 ) )
);
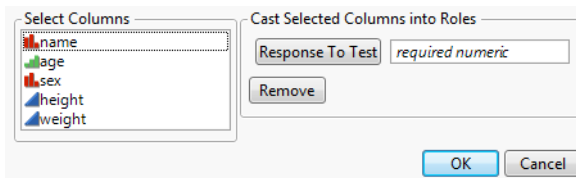
// returns dlg = {x = {:weight}, y = {:height}, Button(1)};
Show( dlg );

If( dlg["Button"] == 1,
  xCol = dlg["x"];
  yCol = dlg["y"];
);

Show( xCol, yCol );
  xCol = {:height};
  yCol = {:weight};
```

`New Window()` is more flexible because of numerous optional arguments. In the following example, the user must select two columns before clicking the OK button. A Bivariate graph is then created. Validating the user's selections is not possible in `Column Dialog()`.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
xvar = .;
yvar = .;
win = New Window( "Example of Returning Values",
  <<Modal,

  // require the user to select two variables before clicking OK
  <<On Validate(
    Show( xvar, yvar );
      If( Is Missing( xvar ) | Is Missing( yvar ),

      // if xvar or yvar are missing do nothing when OK is clicked
      0,
      1
    );
  ),
  Text Box( " Select two numeric columns. " ),
  H List Box(
```

```
        Text Box( " X, Factor " ),
        x = Col List Box(
           dt, // data table reference
           all, // display all columns from the data table

           // get the name of the selected column before the window closes
           xvar = (x << Get Selected)[1];
           Show( xvar );
        ),
        Text Box( "Y, Response" ),
        y = Col List Box(
           dt,
           all,
           yvar = (y << Get Selected)[1];
           Show( yvar );
        )
     )
);
If (win["Button"] == 1, // if the user clicks OK...
xcol = Column( dt, xvar ); //  get the columns
ycol = Column( dt, yvar );
dt << Bivariate( Y( ycol ), X( xcol ) ); // create a Bivariate plot
);
```

Constructors for Column Dialogs

Constructors such as Edit Number() and Check Box() can be used in column dialogs, which
construct launch windows with a column selector. The following example shows a typical
column dialog with an Edit Number() box:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column Dialog(
   ex x = ColList( "X", Max Col( 1 ) ),
   HList( "Alpha",
   ex = Edit Number( .05 ) )
);
```

Table 11.3 describes the column dialog constructors.

Notes:

• A column dialog automatically includes the list of columns in the data table.

• A column dialog also automatically includes OK, Cancel, and Remove buttons.

Table 11.3 Column Dialog Constructors

Constructor	Syntax	Explanation
Button	Button("OK"), Button("Cancel")	Draws an OK or a Cancel button. If OK is clicked, Button(1) is returned. If Cancel is clicked, Button(−1) is returned.
Check Box	*var*(CheckBox("*Text after box*", <1|0>))	When OK is clicked, a selected check box assigns 1 to the variable. A check box that is not selected assigns 0 to the variable. Add an optional 1 for the check box to be selected (on), or 0 for it to be not selected (off) when the window first appears. The default value is 0 (off).
Col List	*var*=Col List("*role*", <MaxCol(*n*)>, <Datatype(*type*)>)	Creates a selection destination with a *role* button; the user's choices are returned in a list item of the form *var*={*choice 1, choice 2, …, choice n*}. You can specify the minimum and maximum number of columns using MinCol(*n*) or MaxCol(*n*). You can specify the required data type of the column using Datatype(*type*). The choices for *type* are Numeric, Character, or Rowstate.
Combo Box	*var*(ComboBox("*choice1*", "*choice2*", ...))	Produces a menu with the choices listed. The first choice is the default. Choices evaluate to quoted text strings. Choices can also be inside a list.
Edit Number	*var*(Edit Number(*number*))	Produces an edit field for a number with *number* as the default value. When OK is clicked, the number entered in the field is assigned to the variable.

Table 11.3 Column Dialog Constructors *(Continued)*

Constructor	Syntax	Explanation
Edit Text	*var*(Edit Text("*string*", <width(*x*)>))	Produces an edit field for a string with *string* as the default value. You can also specify the minimum width of the box in pixels. The default width is 72 pixels. When OK is clicked, the text entered in the field is assigned to the variable.
HList	HList(*item*, *item*, ...)	Top aligns and spaces the *items* in a horizontal row. Placing a pair of VLists within an HList produces a top aligned, spaced pair of columns.
Line Up Box	Line Up Box(*n*, *item_11*, *item_12*, ..., *item_1n*, ..., *item_nn*)	Lines up the *items* listed in *n* columns, where *item_ij* is the *j*th item of the *i*th row.
List Box	var=List Box({"item", "item", ...}, width(50), max selected(2), NLines(6))	Creates a display box that contains a list of items.
Radio Buttons	*var*(RadioButtons("*choice1*", "*choice2*", ...))	Produces a vertical, left-justified list of radio buttons with the choices specified. The first choice is the default. When OK is clicked, the button that is selected is assigned to the variable. Choices must evaluate to quoted text strings.
string	"*string*"	Draws text in the window. For example, can include a labeling string before an Edit Number field. All strings must be quoted.
VList	VList(*item*, *item*, ...)	Left aligns and spaces the *items* in a vertical column. Placing a pair of HLists within a VList produces a left-aligned, spaced pair of rows.

Convert Deprecated Dialog to New Window

Because New Window() provides more display options and better control over the content and functions of the window, rewrite your Dialog() scripts.

Note: Dialog() scripts will not run in the future.

The following sections illustrate how to rewrite Dialog() scripts as New Window() scripts.

New Window Example

The following example creates a window with multiple vertical and horizontal list boxes:

```
win = New Window( "New Window Example",
  <<Modal,
  <<ReturnResult, // ensure that you get the same result as with Dialog
  V List Box(
    V List Box(
      Text Box( "Radio Frequency Embolism Projection" ),
      Line Up Box(
        NCol( 2 ),
        Text Box( "Lower Spec Limit" ),
        lsl_box = Number Edit Box( 230 ),
        Text Box( "Upper Spec Limit" ),
        usl_box = Number Edit Box( 340 ),
        Text Box( "Threshold" ),
        threshold_box = Number Edit Box( 275 )
      ),
      H List Box(
        Panel Box( "Type of Radio",
              rb_box1 = Radio Box( {"RCA", "Matsushita",
              "Zenith", "Sony"} )),
        Panel Box( "Type of Antenna" ,
              rb_box2 = Radio Box( {"Dish", "Helical", "Polarizing",
              "Radiant Array"} ) ) )
      ),
      cb_box1 = Check Box( "Emission Synchronization" ),
      Text Box( "Title for plot" ),
      title_box = Text Edit Box( "My projection" ),
```

```
                H List Box( Text Box( "Quality" ),
                cb_box2 = Combo Box( {"Fealty", "Loyalty", "Piety", "Obsequiousness"}
        ) )
            ),
        H List Box(
            Align( Right ),
            Spacer Box(),
            Button Box( "OK",
                lsl = lsl_box << Get;
                usl = usl_box << Get;
                threshold = threshold_box << Get;
                radio_type = rb_box1 << Get;
                antenna = rb_box2 << Get;
                synch = cb_box1 << Get;
                title = title_box << Get Text;
                quality = cb_box2 << Get;
            ),
            Button Box( "Cancel" )
        )
    )
);
If(win["Button"] == 1,

// if the user clicks OK, show the selections in the log
    Show( "OK", lsl, usl, threshold, radio_type, antenna, synch, title,
    quality);
    ,
    Show( "Canceled" ); // if the user clicked Cancel, print "Canceled"
);
```

Example of Deprecated Dialog

Compare the previous New Window() example to this example of the deprecated Dialog().

```
dlg = Dialog(
    Title( "Dialog Example" ),
    H List(
        V List(
            "Radio Frequency Embolism Projection",
            Lineup( 2,
                "Lower Spec Limit", lsl = Edit Number( 230 ),
                "Upper Spec Limit", usl = Edit Number( 340 ),
                "Threshold", threshold = Edit Number( 275 )
            ),
            H List(
                V List(
```

```
                    "Type of Radio",
                    type = Radio Buttons( "RCA", "Matsushita", "Zenith", "Sony" )
                ),
                V List(
                    "Type of Antenna",
                    antenna = Radio Buttons( "Dish", "Helical", "Polarizing",
                        "Radiant Array" )
                )
            ),
            synch = Check Box( "Emission Synchronization", 0 ),
            "Title for plot",
            title = Edit Text( "My projection" ),
            H List(
                "Quality",
                quality = Combo Box( "Fealty", "Loyalty", "Piety", "Obsequiousness"
                )
            )
        ),
        V List( Button( "OK" ), Button( "Cancel" ) )
    )
);
If( dlg["Button"] == 1,
    Show(
        "OK",
        dlg["lsl"],
        dlg["usl"],
        dlg["threshold"],
        dlg["type"],
        dlg["antenna"],
        dlg["synch"],
        dlg["title"],
        dlg["quality"]
    ),
    Show( "Canceled" )
);
```

Figure 11.38 Results from New Window (left) and the Deprecated Dialog (right)

Note: Use `Return Result` to get the same result in `New Window()` as with the deprecated `Dialog()`. See "Construct a Modal Window" on page 609 for an example.

Differences Between New Window and the Deprecated Dialog

This section shows the differences between display boxes in `New Window()` and the deprecated `Dialog()` scripts. Table 11.4 summarizes those differences. More information can be found in the sections that follow the table.

Table 11.4 Display Boxes in New Window and Deprecated Dialog Scripts

Box Type	New Window	Deprecated Dialog

Strings and Lists	`win = New Window( "Combo Box",`
	`    <<Modal,`
	`    <<ReturnResult,`
	`    cb = Combo Box( {"True",`
	`"False"} ),`
	`);`
		`Dialog( Title( "Combo Box" ),`
		`    cb = Combo Box( "True",`
		`"False" ),`
		`);`

List Boxes and Check Boxes	`win = New Window( "H List Box",`
	`    <<Modal,`
	`    <<ReturnResult,`
	`    H List Box(`
	`        kb1 = Check Box( "a" ),`
	`        kb2 = Check Box( "b" ),`
	`        kb3 = Check Box( "c" )`
	`    ),`
	`);`
		`dlg = Dialog( Title( "H List" ),`
		`    H List(`
		`        kb1 = Check Box( "a", 0 ),`
		`        kb2 = Check Box( "b", 0 ),`
		`        kb3 = Check Box( "c", 0 )`
		`    ),`
		`);`

Table 11.4 Display Boxes in New Window and Deprecated Dialog Scripts *(Continued)*

Box Type	New Window	Deprecated Dialog
| Line Up Items | ```
win = New Window( "Line Up Box",
    <<Modal,
    <<ReturnResult,
    V List Box(
        Line Up Box(NCol(2),
        Text Box("Set this value"),
var1=Number Edit Box(42),
        Text Box("Set another
value"), var2=Number Edit
Box(86),
        ),
    ),
    H List Box(
    Button Box( "OK" ),
    Button Box( "Cancel" )
);
// returns: {Button( 1 )}
``` | ```
dlg = Dialog(
    V List(
        Line Up(2,
        "Set this value",
variable=Edit Number(42),
        "Set another value",
var2=Edit Number(86)),
        H List(Button("OK"),
Button("Cancel")))));
// returns: {variable = 42, var2
= 86, Button(1)}
``` |
| Radio Boxes | ```
win = New Window( "Radio Box",
    <<Modal,
    <<ReturnResult,
    Panel Box( "Select",
        rbox = Radio Box( {"a",
"b", "c"} )
    )
);
``` | ```
dlg = Dialog( Title("Radio
Buttons"),
    rb = Radio Buttons( {"a", "b",
"c"} ),
);
``` |
| Text Edit Boxes | ```
win = New Window( "Text Edit
Box",
    <<Modal,
    <<ReturnResult,
    V List Box(
        Text Box( "Strings" ),
        str1 = Text Edit Box( "The"
),
        str2 = Text Edit Box(
"quick" ),
    )
);
``` | ```
dlg = Dialog(
    Title( "Edit Text" ),
    V List(
        "Strings",
        str1 = Edit Text( "The" ),
        str2 = Edit Text( "quick"
), )
);
``` |

Table 11.4 Display Boxes in New Window and Deprecated Dialog Scripts *(Continued)*

Box Type	New Window	Deprecated Dialog
Number Edit Boxes	```win = New Window("Number Edit Box", <<Modal, <<ReturnResult, V List Box(Text Box("Random Numbers"), num1 = Number Edit Box(Random Uniform()), num2 = Number Edit Box(Random Uniform() * 10), ...);```	```dlg = Dialog(Title("Edit Number"), V List("Random Numbers", num1 = Edit Number(Random Uniform()), num2 = Edit Number(Random Uniform() * 10), ...);```

Strings and Lists

A major difference between the New Window() and the deprecated Dialog() functions is how you specify lists and strings as arguments. In New Window(), the items must be placed in a list. In the deprecated Dialog(), items are generally separated by commas. For example, compare the following two instances of a combo box.

```
win = New Window( "Combo Box", // modal New Window
   <<Modal,
   <<Return Result,
   cb = Combo Box( {"True", "False"} ), // items are in a bracketed list
);

Dialog( Title( "Combo Box" ),// modal Dialog
   cb = Combo Box( "True", "False" ), // items are separated by commas
   );
```

Note: The script does not explicitly define the OK button. For modal windows, JMP automatically adds the OK button. Explicitly define each additional button.

Additionally, the deprecated Dialog() could include " " to indicate empty text. New Window() requires that empty text appear in a Text Box (" ").

List Boxes and Check Boxes

In New Window(), boxes are horizontally aligned with H List Box() and vertically aligned with V List Box(). In the deprecated Dialog(), you use H List and V List instead.

Note: Boxes are vertically aligned by default if H List Box() or V List Box() are omitted.

The following example creates a window with three horizontal check boxes and an OK button:

```
win = New Window( "H List Box",
   <<Modal,
   <<Return Result,
   H List Box(
      kb1 = Check Box( "a" ),
      kb2 = Check Box( "b" ),
      kb3 = Check Box( "c" )
   ),
);
```

Here is the same window created with the deprecated Dialog() and an H List():

```
dlg = Dialog( Title( "H List" ),
   H List(
         kb1 = Check Box( "a", 0 ),
         kb2 = Check Box( "b", 0 ),
         kb3 = Check Box( "c", 0 )
      ),
);
```

Line Up Items

In New Window(), use Line Up Box() to arrange items in the number of columns that you specify. In the deprecated Dialog(), you used Line Up.

The following example arranges the text boxes in one column and the number edit boxes in another column:

```
win = New Window( "Line Up Box",
   <<Modal,
   <<Return Result,
   V List Box(
      Lineup Box( N Col( 2 ),
         Text Box( "Set this value" ),
         var1 = Number Edit Box( 42 ),
         Text Box( "Set another value" ),
         var2 = Number Edit Box( 86 ),

      ),
   ),
   H List Box( Button Box( "OK" ), Button Box( "Cancel" ) ) )
);
```

Clicking **OK** returns the following result in the log window:

```
List( 3 elements ) assigned.
```

Figure 11.39 Default Dialog Arrangements for Windows (left) and macOS (right)

Here's the same window created with the deprecated `Dialog()` and `Line Up`:

```
dlg = Dialog(
    V List(
        Line Up( 2,
            "Set this value", variable = Edit Number( 42 ),
            "Set another value", var2 = Edit Number( 86 )
        ),
        H List( Button( "OK" ), Button( "Cancel" ) )
    )
);
```

Clicking **OK** returns the following result in the log window:

```
{variable = 42, var2 = 86, Button(1)}
```

Note: JMP does exert some control over OK and Cancel button positions to ensure that dialog boxes are consistent with what the operating system expects. In certain cases, JMP needs to override your `H List Box()`, `V List Box()`, and `Line Up Box()` settings for `Button( "OK" )` and `Button( "Cancel" )`. Do not be alarmed if the result is slightly different from what you expect.

Radio Boxes

Another difference between `New Window()` and the deprecated `Dialog()` is the usage of `Radio Box`.

In `New Window()`, you must define the `Panel Box()` container for `Radio Box()` if you want the radio buttons in a box.

```
win = New Window( "Radio Box",
    <<Modal,
    <<Return Result,
    Panel Box( "Select",
        rbox = Radio Box( {"a", "b", "c"} )
    )
);
```

In the deprecated `Dialog()`, the `Radio Buttons()` automatically appear in a panel box.

```
dlg = Dialog( Title( "Radio Buttons" ),
  rb = Radio Buttons( {"a", "b", "c"} ),
);
```

Text Edit Boxes

In `New Window()`, use `Text Edit Box()` to create editable boxes that contain specified strings. In the deprecated `Dialog()`, you used `Edit Text()`.

```
win = New Window( "Text Edit Box",
  <<Modal,
  <<Return Result,
  V List Box(
    Text Box( "Strings" ),
    str1 = Text Edit Box( "The" ),
    str2 = Text Edit Box( "quick" ),
    str3 = Text Edit Box( "brown" ),
    str4 = Text Edit Box( "fox" ),
    str5 = Text Edit Box( "jumps" ),
    str6 = Text Edit Box( "over" ),
    str7 = Text Edit Box( "the" ),
    str8 = Text Edit Box( "lazy" ),
    str9 = Text Edit Box( "dog" ) ) );
```

Here is the same window created with the deprecated `Dialog()` and `Edit Text()`:

```
dlg = Dialog(
  Title( "Edit Text" ),
  V List(
    "Strings",
    str1 = Edit Text( "The" ),
    str2 = Edit Text( "quick" ),
    str3 = Edit Text( "brown" ),
    str4 = Edit Text( "fox" ),
    str5 = Edit Text( "jumps" ),
    str6 = Edit Text( "over" ),
    str7 = Edit Text( "the" ),
    str8 = Edit Text( "lazy" ),
    str9 = Edit Text( "dog" )
  )
);
```

Alternatively, use `String Col Edit Box()` to create editable boxes within in a table structure that contains specified strings. For example, the following script creates a window named "String Col Edit Box". The window contains a column (labeled "Strings") of editable boxes, each of which contains the specified string:

```
win = New Window( "String Col Edit Box",
    <<Modal,
    <<Return Result,
    steb = String Col Edit Box(
        "Strings",
        {"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}
    )
);
```

Note: In the preceding example, `New Window()` assigns only two elements to the list (the specified list items and the button).

Number Edit Boxes

In `New Window()`, use `Number Edit Box()` to create editable boxes that contains the specified numbers. In the deprecated `Dialog()`, you used `Edit Number()`.

```
win = New Window( "Number Edit Box",
    <<Modal,
    <<ReturnResult,
    V List Box(
        Text Box( "Random Numbers" ),
        num1 = Number Edit Box( Random Uniform() ),
        num2 = Number Edit Box( Random Uniform() * 10 ),
        num3 = Number Edit Box( Random Uniform() * 100 ),
        num4 = Number Edit Box( Random Uniform() * 1000 )
    ),
);
```

Here is the same window created with the deprecated `Dialog()` and `Edit Number()`:

```
dlg = Dialog(
    Title( "Edit Number" ),
    V List(
        "Random Numbers",
        num1 = Edit Number( Random Uniform() ),
        num2 = Edit Number( Random Uniform() * 10 ),
        num3 = Edit Number( Random Uniform() * 100 ),
        num4 = Edit Number( Random Uniform() * 1000 )
    ),
);
```

Another method for creating the same window uses Number Col Edit Box() to create editable boxes within in a table structure that contains specified numbers. For example, the following script creates a window named "Number Col Edit Box". The window contains a column (labeled "Random Numbers") of editable boxes, each of which shows the specified type of random number:

```
win = New Window( "Number Col Edit Box",
    <<Modal,
    <<ReturnResult,
    nceb = Number Col Edit Box(
        "Random Numbers",
        {num1 = Random Uniform(), num2 = Random Uniform() * 10, num3 =
        Random Uniform() * 100, num4 = Random Uniform() * 1000}
    ),
);
```

Note: In the preceding example, the Number Col Edit Box() assigns only two elements to the list. The New Window() and deprecated Dialog() examples for Number Col Edit Box() assign five elements to the list.

Optional Scripts in New Window

One of the benefits of using New Window() is the ability to add optional scripts to display boxes. In the deprecated Dialog(), the following combo box could not include an optional script. If you wanted an action associated with any of the display box controls, you had to place a script as the control's last argument. For example:

```
win = New Window( "Combo Box",
<<Modal,
<<ReturnResult,
    comboObj = Combo Box(
        {"True", "False"},
        << Set( 1 ),
        Print( comboObj << Get )
    )
);
```

When the user selects a different value, the selected item number (1 or 2 in this case, because there are two items in the combo box) is printed to the log.

Scripting the Script Editor

Even the script editor window is a display tree in JMP, which means you can write a JSL script to write and save another JSL script.

There is no `New Script` command. Instead, to open a new script window, you use the `New Window()` function and then send it a message to tell it that it's a script window:

```
ww = New Window( "Window Title", <<Script, "Initial Contents" );
```

The last argument is optional. If you include a string, the new script window contains that string.

In the `New Window` example above, `ww` is a reference to the display box that is the entire window. To write to a script window, you first need to get a reference to the part of the display box that you can write to, which is called a script box:

```
ed = ww[Script Box(1)];
```

Using the reference `ed`, you can add text, remove text, and get the text that is already there.

```
ed << Get Text();
    "Initial Contents"
```

Use `Set Text` to set all the text in the script window. The following command clears all text in the Script Window and then adds `aaa=3;` followed by a return:

```
ed << Set Text( "aaa=3;\!N" );
```

Use `Append` to add additional text to the end of the script window.

```
ed << Append Text( "bbb=1/10;" );
ed << Append Text( "\!Nccc=4/100;" );
```

Use the `Get Line Text` command to get the text at the line of a specified line number. Use the `Set Line Text` command to replace a specified line of text with new text.

```
ed << Get Line Text( 2 );
ed << Set Line Text( 2, "bbb = 0.1;" );
```

Use the `Get Line Count` message to get the total number of lines in the script. The `Get Lines` message returns a list of each line in the script as a string.

```
ed << Get Line Count();
ed << Get Lines();
```

Use the `Reformat` message to automatically format a script for easier reading.

```
ed << Reformat();
```

To run an entire script in a script window, use it the `Run` message.

```
ed << Run();
```

To close the script window without saving the script, send the window the `Close Window` message, just like you can do with any JMP window.

```
ww << Close Window( nosave );
```

To save and view the script, use the `Save Text` and `Load Text File` messages:

```
ww = New Window( "Test", <<Script, "Open(\!"$SAMPLE_DATA\Big
  Class.jmp\!");");
ww << Save Text( "$TEMP/Test.jsl" );
Write( Load Text File( "$TEMP/Test.jsl" ) );
```

Note: When the script is loaded into the script window, the window is considered "dirty" or modified. Use `<<Set Dirty(0)` to remove this property. Otherwise, you are prompted to save the script when you close the window.

To save the script and close the window, follow this example:

```
ww = New Window( "Test", <<Script, "Open(\!"$SAMPLE_DATA\Big
  Class.jmp\!");");
ww << Save Text( "$TEMP/Test.jsl" );
ww << Close Window();
```

Technical Details

Write Tab Box and Tab Page Box Scripts

`Tab Page Box()` organizes the title and contents of the page in one display box. When `Tab Page Box()` is inside `Tab Box()`, a tabbed window with multiple tabbed pages appears. See "Tab Box and Tab Page Box" on page 569.

In previous versions of JMP, display boxes such as `V List Box()` contained tab contents. Consider the following script:

```
win = New Window( "Tab Box",
  tb = Tab Box(
    "First page", // name of the tab
    V List Box( // tab contents
      Text Box( "first line of first page" ),
      Text Box( "second line of first page" )
    ),
    "Second page", // name of the tab
    V List Box( // tab contents
      Text Box( "first line of second page" ),
      Text Box( "second line of second page" )
    ),
```

```
      )
   );
```

We recommend that Tab Box() contain a Tab Page Box() for each tab. The preceding
example is rewritten as follows:

```
win = New Window( "Tab Box",
   tb = Tab Box(
      Tab Page Box( // tab contents
         Title( "First page of my tab box" ), // name of the tab
         Text Box( "first line of first page" ),
         Text Box( "second line of first page" )
      ),
      Tab Page Box( // tab contents
         Title( "Second page" ), // name of the tab
         Text Box( "first line of second page" ),
         Text Box( "second line of second page" )
      )
   )
);
```

Note: Tab Page Box() requires that the title be defined in a Title() function.

When a tab page box is created or dragged outside a tab box, the tab page box is a stand-alone
container. The title appears in a shaded box at the top of the page, similar to the title in Sheet
Part(); the title does not appear in an interactive tab. "Dashboard with One Row of Reports"
on page 596 provides an example.

Deprecated Tab Box and Tab Page Box Messages

Some messages previously used for Tab Box() are deprecated. Their counterparts are
compatible with Tab Page Box().

Table 11.5 Deprecated Tab Box() Messages

Deprecated Tab Box Message	Current Tab Page Box Message
Set Title	Title
Set Tip	Tip
Set Icon	Icon
Set Closeable	Closeable

See "Tab Box and Tab Page Box" on page 569 for more information about the two display
boxes.

Chapter **12**

Scripting Graphs

Editing and Creating Two-Dimensional Graphs

The JSL script for a graph contains expressions that create shapes and lines, apply fill patterns, assign background colors, and so on. You can also right-click a graph and select Customize to add the same expressions.

This chapter also describes how to customize hover labels with rich text, images, and drill-down graphs.

For information about scripting three-dimensional graphs, see the chapter "Three-Dimensional Scenes" on page 723.

Contents

Add Scripts to Graphs

When you right-click a graphics frame, you can enter or paste JSL commands as shown in the following example. The script usually contains drawing commands that run in the context of the graphics frame. The context of the graphics frame includes the data range of the axes and the order that the data and scripts are drawn in.

1. Select **Help > Sample Data Library** and open Big Class.jmp.
2. Select **Analyze** > **Fit Y by X**.
3. Choose weight for Y and height for X, and then click **OK**.
4. Right-click inside the graph and select **Customize**.

Figure 12.1 Customizing the Graph Options

5. Click the **Add** button ⊞ to add a new graphics script.
6. Type the following text and click **OK**.

```
Text( {55, 160}, "Hello World" );
```

Now the graph has a text element at the graph's *x*-coordinate 55 and *y*-coordinate 160.

Figure 12.2 Adding a Script to a Graph Interactively

By default, the data points appear on top of graphics.

For example, add a new script in the Customize Graph window:

```
Fill Color( "Green" ); Rect( 57, 175, 65, 110, 1 );
```

A solid green rectangle appears. The list of scripts shows the order in which each script is drawn, so the first item on the list is drawn first.

- If you want the rectangle behind everything, move the rectangle script above the text script: select the rectangle script and click the up button ⬆.

- If you want the rectangle on top of everything, select the rectangle script and click the down button ⬇.

You can arrange all the scripts in a graph into the drawing order that you prefer. Any new script is initially added directly after the item in the list that is selected.

Tip: To use a script that references a column name, use `Column( colname )` or a colon (`:colname`) to scope it properly.

Hint: To see the JSL for the above actions, select **Save Script > To Script Window** from the red triangle menu.

Specify the Order of Graphical Elements

You can also add graphics elements using JSL instead of doing so interactively:

```
Frame Box <<Add Graphics Script(<order>, <"description">, script)
```

When you add graphical elements using JSL, they are drawn on top of whatever is already in the graph.

- The optional `order` argument specifies in what order to draw the graphics element. The order can be the keyword `Back` or `Front` *or* an integer that specifies the drawing order for a number of graphics elements. For example, if you add an oval to a scatterplot, the oval is drawn on top of the markers. The keyword `Back` or `1` cause the oval to be drawn last. `Front` or `2` means the object is drawn first.

- The optional `Description` argument is a string that appears in the Customize Graph window next to the graphics script.

To specify the drawing order for a number of graphics elements, use an integer for the `order` argument to determine where each is drawn in relation to the others. The following script first adds a blue oval and then adds a red oval over the blue oval. Both ovals are behind the points.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
op = dt << Bivariate(
  X( :height ),
  Y( :weight ),
);

Report( op )[Frame Box( 1 )] << Add Graphics Script(
  1, // draw the blue oval
  Description( "Blue Oval" ),
  Fill Color( "Blue" );
  Oval( 60, 140, 65, 90, 1 );
);

Report( op )[Frame Box( 1 )] << Add Graphics Script(
  2, // draw the red oval over the blue oval
  Description( "Red Oval" ),
  Fill Color( "Red" );
  Oval( 50, 120, 65, 100, 1 );
);
```

Figure 12.3 Specifying the Drawing Order

Copy and Paste Frame Contents or Settings

You can copy and paste the contents or the settings of a frame with these JSL commands:

```
obj << Copy Frame Contents // contents such as a graph or line of fit
obj << Paste Frame Contents
obj << Copy Frame Settings // settings such as background color
obj << Paste Frame Settings
```

The following example creates two Bivariate graphs, adds a line of fit to the first graph, and then copies and pastes the line to the second graph.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :weight ), x( :height ) );
rbiv1 = biv << Report; // create a report object
rbiv2 = rbiv1 << Clone Box; // create a copy of the Bivariate graph
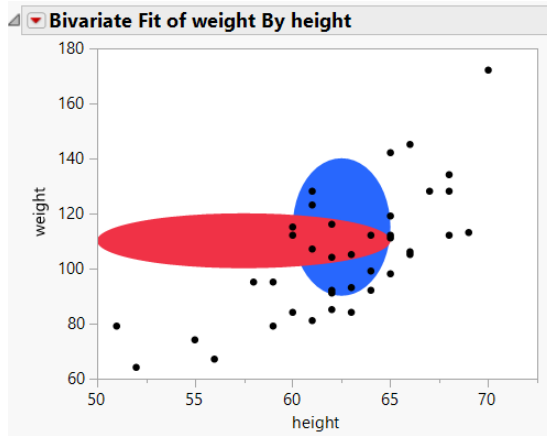
/* place the cloned box below the Bivariate graph
to create a second Bivariate graph */
rbiv1 << Append( rbiv2 );

framebox1 = rbiv1[Frame Box( 1 )]; /* assign references to
the reports' frame boxes*/
framebox2 = rbiv2[Frame Box( 1 )];
biv << Fit Line; // fit a line to the first Bivariate graph

// copy the frame contents from framebox1
framebox1 << Copy Frame Contents;

// paste the frame contents to framebox2
framebox2 << Paste Frame Contents;
```

Figure 12.4 Copying and Pasting Frame Contents

The following example creates two Bivariate graphs, sets the background of the first graph to blue, and then copies and pastes the blue background setting to the second graph:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
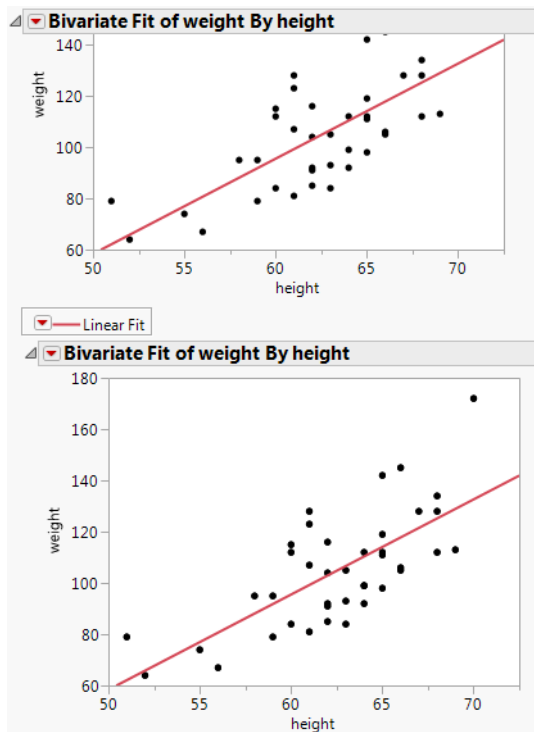
biv1 = dt << Bivariate( y( :weight ), x( :height ) );

// create the first Bivariate graph
rbiv1 = biv1 << Report; // create a report object
rbiv2 = rbiv1 << Clone Box; // create a copy of the Bivariate graph

/* place the cloned box below the Bivariate graph
to create a second Bivariate graph */
rbiv1 << Append( rbiv2 );

framebox1 = rbiv1[Frame Box( 1 )]; /* assign a reference to the
                                     reports' frame boxes */
framebox2 = rbiv2[Frame Box( 1 )];

// assign the background color to framebox1
framebox1 << Background Color( "Blue" );
```

```
// copy the frame settings from framebox1
framebox1 << Copy Frame Settings;

// paste the frame settings to framebox2
framebox2 << Paste Frame Settings;
```

Figure 12.5 Copying and Pasting the Background

Copy and Paste Histograms

The following example shows how to create two histograms and copy the second histogram into the first histogram. A platform will not save a pasted-in segment, so you'll need to run a script like this to reproduce the graph.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dist = dt << Distribution(
    SendToByGroup( {:sex == "F"} ),
    Nominal Distribution( Column( :age ) ),
    Histograms Only,
    By( :sex ),
```

```
SendToByGroup(
    {:sex == "M"},
    SendToReport(
        Dispatch(
            {"Distributions sex=M", "age"},
            "Distrib Nom Hist",
            FrameBox,
            {DispatchSeg(
                Hist Seg( 1 ),
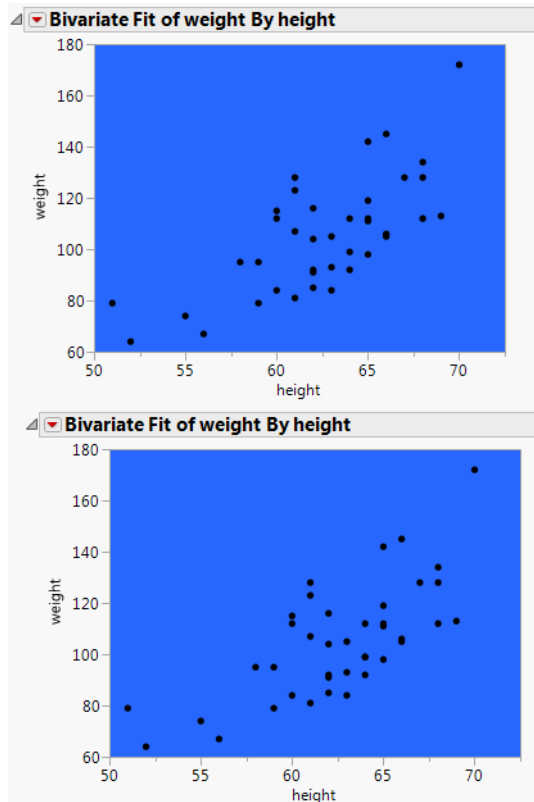
                // set the histogram color for males
                Fill Color( "Light Yellow" ),
            )}
        )
    )
);

For( i = 2, i <= N Items( dist ), i++,
    Report( dist[i] )[FrameBox( 1 )] << Copy Frame Contents;

    // copy the the second histogram into the first histogram
    Report( dist[1] )[FrameBox( 1 )] << Paste Frame Contents;
);

New Window( "Distribution", Outline Box( "age", Report( dist[1] )[Picture Box(
    1 )] ) );
```

Figure 12.6 Pasted Histogram

Interactively Copy and Paste Frame Contents and Settings

To interactively copy and paste the frame settings, follow these steps:

1. Right-click the graph and select **Edit > Copy Frame Settings**.
2. Right-click the frame to which you want to paste the settings and select **Edit > Paste Frame Settings**.

To interactively copy and paste the frame contents, follow these steps:

1. Right-click the graph and select **Edit > Copy Frame Contents**.
2. Right-click the frame to which you want to paste the contents and select **Edit > Paste Frame Contents**.

Create New Graphs from Scratch and Customize Them

Graphics scripts are set up inside the Graph Box() command within a New Window() command.

```
New Window("title", <arguments>, Graph Box( named arguments,..., script));
```

The following arguments are named arguments for Graph Box():

```
Frame Size( horizontal, vertical ), // size in pixels
X Scale( xmin, xmax ), Y Scale( ymin, ymax ), // range of x and y axes
X Name( "x" ), Y Name( "y" ), // names for x and y axes
Suppress Axes // omit the axes
```

For example, Graph Box() accepts named arguments as shown in the following example, which creates a graph with a red background.

```
win = New Window( "Named arguments",
    Graph Box(
        Frame Size( 300, 300 ), // construct the graph box
        X Scale( 0, 190 ), // set the x-axis range
        Y Scale( 0, 190), // set the y-axis range
        X Name( "height" ),
        Y Name( "weight" ),
        <<Background Color( "Red" ) // set the background color
    )
);
```

Figure 12.7 Creating a Graph

Alternatively, you can use the send << operator to send commands to the Graph Box instead of using the named arguments. This example also creates a graph with a red background.

```
win = New Window( "Messages",
    Graph Box(
        <<Frame Size( 300, 300 ),
        <<XAxis( 0, 190 ),
        <<YAxis( 0, 190 ),
        <<Background Color( "Red" )
    )
);
```

Customize Graphs

You can also make changes to graphs in a script. For example, create a window with a graph, get a reference to the report, and then set the frame box size.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :weight ), X( :height ), Fit Line );
rbiv = biv << Report;
rbiv[Frame Box( 1 )] << Frame Size( 400, 400 );
```

To see a list of possible messages for any given display box object, select **Help > Scripting Index** and select **Display Boxes** from the list. Running the Show Properties() command is an alternative to using the Scripting Index. For example, here is a partial list of messages that you can send to an axis:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :weight ), X( :height ), Fit Line );
rbiv = biv << Report;
Show Properties( rbiv[Axis Box( 1 )] );
    Axis Settings [Action](Provides a way to bring up the axis dialog or set
        configurable axis options for a given axis.)
    Revert Axis [Action](Restore the settings that this axis had originally.)
    Add Axis Label [Action]
    Remove Axis Label [Action]
    ...
```

The following expressions change the font of both axis labels (weight and height in the example above) to 12-point, italic Arial Black:

```
rbiv[Text Edit Box( 1 )] << Set Font( "Arial Black", 12, Italic );
rbiv[Text Edit Box( 2 )] << Set Font( "Arial Black", 12, Italic );
```

or

```
rbiv[Text Edit Box( 1 )] << Set Font( "Arial Black" );
rbiv[Text Edit Box( 1 )] << Set Font Style( "Italic" );
rbiv[Text Edit Box( 1 )] << Set Font Size( 12 );
rbiv[Text Edit Box( 1 )] << Set Font( "Arial Black" );
rbiv[Text Edit Box( 2 )] << Set Font Style( "Italic" );
rbiv[Text Edit Box( 2 )] << Set Font Size( 12 );
rbiv[Text Edit Box( 2 )] << Set Font Name( "Arial Black" ) << Set Font Size(
    12 ) << Set Font Style( Italic )
```

The default font specified in the Preferences is applied if the font is not installed on the computer.

Create Shapes for a Bubble Plot

Bubble Plot supports predefined shapes and custom shapes. Shapes can give visual cues about the data or provide a more interesting shape than a circle. For example, you can show arrows (Figure 12.8) or a hurricane symbol on a map (Figure 12.9).

Specify a Predefined Shape

The Set Shape message specifies the predefined shape:

```
bp << Set Shape( "Circle"|"Triangle"|"Square"|"Diamond"|"Arrow" );
```

The following example creates the bubble plot and diamond shape shown in Figure 12.8:

```
dt = Open( "$SAMPLE_DATA/PopAgeGroup.jmp" );
bp = dt << Bubble Plot(
    X( :"Portion 0-19"n ),
    Y( :"Portion60+"n),
    Sizes( :Pop ),
    ID( :Country )
);
bp << Set Shape( "Diamond" );
```

Figure 12.8 Diamond Shapes

Create a Custom Shape

You can create custom shapes only in JSL through a combination of two messages:

```
bp << Set Custom Path( pathMatrix | pathText );
bp << Set Shape( "Custom" );
```

- In Set Custom Path(), the path can consist of an *N*x3 matrix or a string that contains SVG code. The default value is 0.

```
bp << Set Custom Path(
    "M-1,-1 L-1,1 L0,0.5 L1,1 L1,-1 L0,-0.5 L-1,-1 Z, ..." );
```

- In Set Shape(), specify "Custom".

```
bp << Set Shape( "Custom" );
```

The following script creates the map and custom hurricane shape shown in Figure 12.9.

```
dt = Open( "$SAMPLE_DATA/Hurricanes.jmp" );
```

```
dt << Bubble Plot(
    X( :Longitude ),
    Y( :Latitude ),
    Sizes( "Wind (Knots)"n ),
    Time( :Date ),
    Coloring( :Landfall in USA ),
    ID( :Name and ID ),
    Speed( 220 ),
    Bubble Size( 51.43 ),
    Time Index( 729.9 ),
    Trail Bubbles( 1 ),
    All Labels( 0 ),
    No Labels( 0 ),
/* rotate shapes in the direction of the movement
    good for animated plots */
    Orient Shapes( 1 ),
    Set Custom Path( // specify the SVG string
        "M-0.99999898,0.00921400000000006 C-0.99999898,-0.51266322
    -0.55204076,-0.93622284 -0.000119060000000104,-0.93622284
    C0.1835362,-0.93622284 0.3556076,-0.88949112 0.5035232,-0.80766502
    C0.3236248,-1.36413554 0.0260398000000001,-1.815694896 -0.3555946,-2
    C0.4067376,-1.7567781 0.9372286,-0.91116206 1.0000004,0.0082972000000009
    C1.0000004,0.5301742 0.5518026,0.9548002 -0.00011905999999982,0.9547998
    C-0.18139054,0.9547998 -0.35150496,0.9090838 -0.49816916,0.8292164
    C-0.31759686,1.3771358 -0.0174124399999999,1.8176124 0.3602494,1.999999
    C-0.38991788,1.7606584 -0.92375834,0.908779 -0.99999898,0.00921400000000034
    z"
    ),
    Set Shape( "Custom" ), // set the custom shape
    Title Position( -88.679, 59.73 ),
    SendToReport(
        Dispatch(
            {},
            "Bubble Plot",
            FrameBox,
            {Frame Size( 743, 603 ), Background Map( Images( "Simple Earth" ) ),
            Grid Line Order( 3 ), Reference Line Order( 2 )}
        )
    )
);
```

Figure 12.9 Custom Hurricane Shape

How to Find Shapes

Search Wikimedia Commons or Google Images and include the keyword *SVG*. Look for SVG examples that are composed of stroke paths. SVG can be used to represent very complex drawings, with gradients and other elements. Only a path representation, which provides both outline and fill modes, is supported.

You might need to translate and scale the SVG that you find in order to get correct coordinates. Inkscape or similar tools can be useful for doing this manipulation. To copy the path from Inkscape, select **Edit > XML Editor**, find the svg:path element, and copy the path string from the d Attribute into Set Custom Path().

Add Lines, Arrows, Points, Shapes, and Text

- "Draw Lines"
- "Draw Arrows"
- "Draw Markers"
- "Draw Pies and Arcs"
- "Draw Regular Shapes: Circles, Rectangles, and Ovals"
- "Draw Irregular Shapes: Polygons and Contours"
- "Add Text"
- "Specify Colors"
- "Specify Transparency"
- "Add Fill Patterns"
- "Specify Line Types"
- "Draw with Pixels"

Draw Lines

Line() draws lines between points.

```
win = New Window( "Five-Point Star",
  Graph Box(
    Frame Size( 300, 300 ),
    X Scale( -1.1, 1.1 ),
    Y Scale( -1.1, 1.1 ),
    Line(
      {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )},
      {Cos( 9 * Pi() / 10 ), Sin( 9 * Pi() / 10 )},
      {Cos( 17 * Pi() / 10 ), Sin( 17 * Pi() / 10 )},
      {Cos( 5 * Pi() / 10 ), Sin( 5 * Pi() / 10 )},
      {Cos( 13 * Pi() / 10 ), Sin( 13 * Pi() / 10 )},
      {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )}
    )
  )
);
```

Figure 12.10 Using Lines to Draw a Star

You can either specify the points in two-item lists as demonstrated above or as matrices of *x* and then *y* coordinates. Matrices are flattened by rows. You can use either row or column vectors, as long as you have the same number of elements in each matrix. The following expressions have the same effect:

```
Line( {1,2}, {3,0}, {2,4} ); // several {x,y} lists
Line( [1 3 2],[2 0 4] ); // row vectors
Line( [1,3,2], [2,0,4] ); // column vectors
Line( [1 3 2], [2,0,4] ); // one of each
```

The star example could also be drawn this way. Note that the script must include the full Matrix({...}) notation rather than [ ] shorthand because the entries are expressions. The following example uses the Matrix() function.

```
win = New Window( "Five-Point Star",
  Graph Box(
    Frame Size( 300, 300 ),
    X Scale( -1.1, 1.1 ),
    Y Scale( -1.1, 1.1 ),
    Line(
      Matrix(
        { // the x coordinates
        Cos( 1 * Pi() / 10 ), Cos( 9 * Pi() / 10 ), Cos( 17 * Pi() / 10 ),
        Cos( 5 * Pi() / 10 ), Cos( 13 * Pi() / 10 ), Cos( 1 * Pi() / 10 )}
      ),
      Matrix(
        { // the y coordinates
        Sin( 1 * Pi() / 10 ), Sin( 9 * Pi() / 10 ), Sin( 17 * Pi() / 10 ),
        Sin( 5 * Pi() / 10 ), Sin( 13 * Pi() / 10 ), Sin( 1 * Pi() / 10 )}
      )
    ) ) );
```

`HLine()` draws a horizontal line across the graph at the *y*-value that you specify. Similarly, `VLine()` draws a vertical line down the graph at the *x*-value that you specify. Both functions support drawing multiple lines by using a matrix of values in the *y* argument. See "Mousetrap()" on page 674.

Draw Arrows

`Arrow()` draws an arrow from the first point to the second point. The default arrowhead is scaled to 1 plus the square root of the length of the arrow. To set the length of the arrowhead, add an optional first argument, specifying the length of the arrowhead in pixels. The following example draws simple arrows.

```
win = New Window( "Arrowheads",
  Graph Box(
    Pen Size( 4 );

    /* matrix 1 defines the x coordinates of an arrow
    matrix 2 defines the y coordinates of an arrow */
    Arrow( 20, [10 30 90], [88 22 44] );
  )
);
```

The following example draws arrows in a circular pattern.

```
win = New Window( "Hurricane",
  Graph Box(
    Frame Size( 100, 100 ),
    X Scale( -100, 100 ),
    Y Scale( -100, 100 ),
    For( r = 35, r < 100, r += 20,
      ainc = 2 * Pi() * 3 / r;
      For( a = 0, a < 2 * Pi(), a += ainc,
        x = r * Cosine( a );
        y = r * Sine( a );
        aa = a + ainc * 45 / r;
        rr = r - r / 6;
        x2 = rr * Cosine( aa );
        y2 = rr * Sine( aa );

        /* list 1 defines the start point for the arrow
        list 2 defines the end point for the arrow */
        Arrow( {x, y}, {x2, y2} );
      );
    )
  )
);
```

Figure 12.11 Drawing Arrows

The following example compares drawing with a specified length (19 pixels) and drawing with the default arrow head size.

```
win = New Window( "Arrowheads",
    Graph Box(
        Frame Size( 300, 300 ),
        X Scale( 0, 100 ),
        Y Scale( 0, 220 ),
        x = 10;
        y1 = 10;
        y2 = y1 + 10;
        For( i = 1, i < 10, i++,
            Pen Color( "Red" );
            Arrow( {x, y1}, {x, y2} );
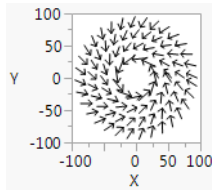
            y2 += 10;
            y1 += 100;
            y2 += 100;
            Pen Color( "Blue" );
            Arrow( 20, {x, y1}, {x, y2} );

            x += 10;
            y1 -= 100;
            y2 -= 100;

            Text Color( "Red" );
            Text( {10, 80}, "Without Length Arg" );

            Text Color( "Blue" );
            Text( {10, 200}, "With Length Arg" );
        );
    )
);
```

Figure 12.12 Arrowhead Sizes

As with `Line()`, you can either specify the points in two-item lists as demonstrated above or as matrices of *x* and then *y* coordinates.

Draw Markers

`Marker()` draws a marker of the type that you specify (1–20) in the first argument at the point that you specify in the second argument. `Marker Size()` scales markers from 0 to 6 (dot to XXXL). To set markers to the size that is specified in the Preferences, use a value of −1.

```
ymax = 20;
win = New Window( "Markers",
   Graph Box(
      Frame Size( 300, 400 ),
      X Scale( -2, ymax - 5 ),
      Y Scale( -2, ymax + 3 ),
      For( j = 1, j < 7, j++,
         Marker Size( j );
         For( i = 0, i < (ymax + 1), i++,
            Marker( i, {j * 2, i} );
            Text( {0, i}, i );
            Text( {j * 2, ymax + 2}, j );
         );
      )
   )
);
```

Figure 12.13 Drawing Markers

You can also include a row state argument before, after, or instead of the marker ID argument. By using Combine States(), you can set multiple row states inside Marker(). Try substituting each of the following lines in the preceding script:

```
Marker( i, Color State( i ), {j * 2, i} );
Marker( Color State( i ), i, {j * 2, i} );
Marker(
    Combine States( Color State( i ), Marker State( i ), Hidden State( i ) ),
    {j * 2, i}
);
```

As with Line() and Arrow(), you can also specify points as matrices of x and then y coordinates.

Draw Pies and Arcs

Pie() and Arc() draw wedges and arc segments. The first four arguments are x1, y1, x2, and y2, the coordinates of the rectangle to inscribe. The last two arguments are the starting and ending angle in degrees, where 0 degrees is 12 o'clock and the arc or slice is drawn clockwise from start to finish.

```
win = New Window( "Pies and Arcs",
    Graph Box(
        Frame Size( 400, 400 ),
        X Scale( 0, 9 ),
        Y Scale( 0, 9 ),
```

```
        Fill Color( "Black" ), // top left
        Pie( 1.1, 7.9, 3.9, 5.1, 45, 270 ),
        Text( Erased, {1.75, 6}, "1,8,4,5,45,270" ),
        Arc( 1, 8, 4, 5, 280, 35 ),
        Fill Color( "Red" ), // top right
        Pie( 7.9, 7.9, 5.1, 5.1, 270, 360 ),
        Text( Erased, {5.75, 6}, "8,8,5,5,270,360" ),
        Arc( 8, 8, 5, 5, 370, 260 ),
        Fill Color( "BlueCyan" ), // bottom left
        Pie( 1.1, 1.1, 3.9, 3.9, 50, 360 ),
        Text( Erased, {1.75, 2}, "1,1,4,4,50,360" ),
        Arc( 1, 1, 4, 4, 370, 40 ),
        Fill Color( "Purple" ), // bottom right
        Pie( 7.9, 1.1, 5.1, 3.9, 270, 45 ),
        Text( Erased, {5.75, 2}, "8,1,5,4,270,45" ),
        Arc( 8, 1, 5, 4, 55, 260 )
    )
);
```

Figure 12.14 Drawing Pies and Arcs

Draw Regular Shapes: Circles, Rectangles, and Ovals

Circles

Circle() draws a circle with the center point and radius given. Subsequent arguments specify additional radii.

```
win = New Window( "Circles",
  Graph Box( Frame Size( 200, 200 ),
    Circle( {50, 50}, 10, 12, 25 )
  )
);
```

Circle() also has a final optional argument, "fill". This string indicates that all circles defined in the function are filled with the current fill color. If "fill" is omitted, the circle is empty.

Figure 12.15 Drawing Circles

Note that a circle is always a circle, even if you resize the graph to a different aspect ratio. If you want your circle to change aspect ratios (in other words, cease being a circle) when the graph is resized, use an oval instead.

If you do not want your circle to resize if the graph is resized, specify the radius in pixels instead:

```
win = New Window( "Circles",
  Graph Box(
    Frame Size( 200, 200 ),
    Circle( {50, 50}, Pixel Radius( 10 ), Pixel Radius( 12 ), Pixel Radius(
  25 ) )
  )
);
```

Rectangles

`Rect()` draws a rectangle from the diagonal coordinates you specify. The coordinates can be specified either as four arguments in order (*left, top, right, bottom*), or as a pair of lists ({*left, top*}, {*right, bottom*}).

```
win = New Window( "Rectangles",
   Graph Box(
      Frame Size( 200, 200 ),
      Pen Color( 1 );
      Rect( 0, 40, 60, 0 );
      Pen Color( 3 );
      Rect( 10, 60, 70, 10 );
      Pen Color( 4 );
      Rect( 50, 90, 90, 50 );
      Pen Color( 5 );
      Rect( 0, 80, 70, 70 );
   )
);
```

`Rect()` has an optional fifth argument, *fill*. Specify 0 to get an unfilled rectangle and 1 to get a filled rectangle. The default value for *fill* is 0. The rectangle is filled with the current `fill color`.

Any negative fill argument produces an unfilled frame inset by one pixel:

```
win = New Window( "Framed Rectangle",
   Graph Box( Frame Size( 200, 200 ), Rect( 0, 40, 60, 0, -1 ) )
);
```

Ovals

`Oval()` draws an oval inside the rectangle given by its *x1, y1, x2*, and *y2* arguments:

```
win = New Window( "Ovals",
   Graph Box(
      Frame Size( 200, 200 ),
      Pen Color( 1 );
      Oval( 0, 40, 60, 0 );
      Pen Color( 3 );
      Oval( 10, 60, 70, 10 );
      Pen Color( 4 );
      Oval( 50, 90, 90, 50 );
      Pen Color( 5 );
      Oval( 0, 80, 70, 70 );
   )
);
```

Oval() also has an optional fifth argument, `fill`. Specify 0 to get an unfilled rectangle and 1 to get a filled oval. The default value for `fill` is 0. The oval is filled with the current `fill` color.

Figure 12.16 shows rectangles and ovals, drawn both filled and unfilled. Notice that filled rectangles do not have outlines. Ovals do have outlines. If you want a filled rectangle with an outline, you must draw the filled rectangle and then draw an unfilled rectangle with the same coordinates.

Figure 12.16 Rectangles and Ovals, Unfilled and Filled

Draw Irregular Shapes: Polygons and Contours

Polygons

Polygon() works similarly to Line(). They both connect points, but Polygon() returns to the first point to close the polygon and filling the resulting area. You can specify the points as individual points in two-item lists (as shown in "Draw Markers" on page 650) or as matrices of *x* and then *y* coordinates. Matrices are flattened by rows, so you can use either row or column vectors, as long as you have the same number of elements in each matrix. First set up the matrices of points and then call them inside Polygon().

```
gCoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75,
    12.5, 6.25, 2.5,
```

```
      1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75,
         40.625, 42.5, 43.125,
      42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50,
         45, 37.5, 32.5,
      28.75, 27.5, 26.25, 25.625, 25];
      gCoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41,
         40, 39, 40, 42, 45,
      50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37, 37, 36, 35,
         30, 25, 20, 15, 10, 5, 2];
      win = New Window( "The JMP Man",
         Graph Box(
            Frame Size( 300, 300 ),
            X Scale( -10, 80 ),
            Y Scale( -10, 80 ),
            Pen Color( "Black" );
            Fill Color( "Blue" );
            Polygon( gCoordX, gCoordY );
            Fill Color( "Black" );
            Circle( {18, 58}, 9, "FILL" );
         )
      );
```

Figure 12.17 Drawing a Polygon

A related command, In Polygon(), tells whether a given point falls inside the specified polygon. This code checks some points from the JMP man pictured in Figure 12.17:

```
      In Polygon( 0,60, GcoordX,GCoordY );  // return 0
      In Polygon( 30,38, GcoordX,GCoordY ); // return 1
```

You can add In Polygon() to the JMP man script. Run the following script, and then click various locations in the picture and watch the log window. "In" and "out" are printed to the log when you click the body.

```
gCoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75,
    12.5, 6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375,
    37.5, 38.75, 40.625, 42.5, 43.125, 42.5, 41.25, 38.75, 43.75, 50, 56.25,
    60.625, 63.75, 65.625, 62.5, 56.25, 50, 45, 37.5, 32.5, 28.75, 27.5, 26.25,
    25.625, 25];
gCoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41,
    40, 39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40,
    38, 36, 37, 37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
win = New Window( "The JMP Man",
    Graph Box(
        Frame Size( 300, 300 ),
        X Scale( -10, 80 ),
        Y Scale( -10, 80 ),
        Pen Color( "Black" );
        Fill Color( "Black" );
        Polygon( gCoordX, gCoordY );
        Mousetrap(
            {},
            Print( If( In Polygon( x, y, gCoordX, gCoordY ), "in", "out" ) )
        );
    )
);
```

Contours

Contour() draws contour lines using a grid of coordinates.

```
Contour( xVector, yVector, zGridMatrix, zContour, <zColors> );
```

Suppose that you have an **n** by **m** matrix *zGridMatrix* of values on some surface (defined across the *n* values of *xVector* by the **m** values of *yVector*). Contour() draws the contour lines that are defined by the values in *zContour* in the colors defined by *zColors*. Here is an example:

```
x = (-10 :: 10) / 5;
y = (-12 :: 12) / 5;
grid = J( 21, 25, 0 );
z = [-.75, -.5, -.25, 0, .25, .5, .75];
zcolor = [3, 4, 5, 6, 7, 8, 9];
For( i = 1, i <= 21, i++,
    For( j = 1, j <= 25, j++,
        grid[i, j] = Sin( (x[i]) ^ 2 + (y[j]) ^ 2 )
    )
);
```

```
Show( grid );
win = New Window( "Hat",
  Graph Box(
    X Scale( -2, 2 ),
    Y Scale( -2.4, 2.4 ),
    Contour( x, y, grid, z, zcolor )
  )
);
```

Figure 12.18 Drawing Contour Lines

Add Text

You can use `Text()` to draw text at a given location.

```
Text( <properties>, ( {x, y}|{left, bottom, right, top} ), "text" );
```

The point and text can be in any order and repeated. You can precede the point and text with an optional first argument: `Center Justified`, `Right Justified`, `Erased`, `Boxed`, `Counterclockwise`, or `Clockwise`. `Erased` is for removing whatever would otherwise obscure the text in a graph. It paints a background-colored rectangle behind the text.

In the following example, notice how the erased text appears inside a white box over the green `Rect()`.

```
mytext = New Window( "Adding Text",
  Graph Box(
    Frame Size( 200, 200 ),
    Y Scale( 0, 15 ),
    X Scale( 0, 10 ),
    Text Size( 9 );
    Text Color( "blue" );
    Text( {5, 1}, "Left Justified" );
    Text( Center Justified, {5, 2}, "Center Justified" );
    Text( Right Justified, {5, 3}, "Right Justified" );
```

```
        Fill Color( 4 );
        Rect( 5, 8, 9, 5, 1 );
        Text( Erased, {6, 6}, "Erased" );
        Text( Boxed, {6, 10}, "Boxed" );
        Text( Clockwise, {4, 10}, "Clockwise" );
        Text( Counterclockwise, {3, 5}, "Counterclockwise" );
    )
);
```

Figure 12.19 Drawing Text in a Graph Box

There is a variant of the `Text()` function that draws a string inside the rectangle that is specified by four coordinates. Here is the syntax:

```
Text( {left, top, right, bottom}, string);
```

Specify Colors

The following commands control colors:

- `Fill Color()` for solid areas
- `Level Color()` for categorical data
- `Pen Color()` for lines and points
- `Back Color()` for the background of text (similar to the box around the erased text in Figure 12.19)
- `Background Color()` for the graph's background color
- `Font Color()` for added text

Fill colors overwrite pen colors for drawn shapes. You do not get both, as in some drawing packages. To get both a fill and a pen line, draw two shapes, one with fill and one without. You can select a color with a single numeric argument, a color name in quotation marks, or an RGB value. The standard colors are specified with numbers 0–15 (both 0 and 15 are black) or by their names (Table 12.1).

Tip: For more shades, add the word Dark, Medium Dark, Medium Light, or Light before a color name in quotation marks.

Table 12.1 Standard JMP Colors

Number	Name
0	Black
1	Gray
2	White
3	Red
4	Green
5	Blue
6	Orange
7	BlueGreen
8	Purple
9	Yellow
10	Cyan
11	Magenta
12	YellowGreen
13	BlueCyan
14	Fuchsia

The following script creates a graph that shows all JMP colors:

```
Text Color( 0 );
New Window( "Colors",
    Graph Box(
        FrameSize( 640, 400 ),
        Y Scale( -1, 17 ),
        X Scale( -3, 12 ),
        k = 0;
        For( jj = 1, jj <= 12, jj += 2,
            l = 15;
```

```
For( i = 0, i <= 15 & k < 85, i++,
    thiscolor = Color To RGB( k );
    Fill Color( k );
    thisfill = 1;
    If( thiscolor == {1, 1, 1},
        Pen Color( 0 );
        thisfill = 0;
    ,
        Pen Color( k )
    );
    Rect( jj, l + .5, jj + .5, l, thisfill );
    Text( {jj - 1, l}, "color ", k );
    k++;
    l--;
    );
);
jj = -2;
color = {"Black", "Gray", "White", "Red", "Green", "Blue", "Orange",
"BlueGreen",
    "Purple", "Yellow", "Cyan", "Magenta", "YellowGreen", "BlueCyan",
"Fuschia", "Black"};
For(
    i = 0;
    l = 15;, i <= 15 & l >= 0,
    i++;
    l--;,
    Text( {jj, l}, color[i + 1] )
);
) );
```

Larger numbers cycle through shading variations of the same color sequence. A script demonstrating this appears under "Colors and Markers" on page 451 in the "Data Tables" chapter. Values outside the range 0–84 are not accepted.

Table 12.2 How Numbers Map to Colors

Number	Result
16–31	Medium Dark
32–47	Medium Light
48–63	Dark
64–79	Light

Table 12.2 How Numbers Map to Colors *(Continued)*

Number	Result
80–84	Light, Medium Light, Gray, Medium Dark, Dark

If you prefer to use RGB values, create a list with the fraction for each color in red, green, and blue order.

```
Pen Color( {.38,.84,.67} ); // teal
```

RGB Color() and Color to RGB() convert color values between JMP color numbers and the red-green-blue system. The following example finds the RGB values for JMP color 3 (red):

```
Color to RGB( 3 );
    {0.941176470588235, 0.196078431372549, 0.274509803921569}
```

Likewise, HLS Color() and Color to HLS() convert color values between JMP color numbers and the hue-lightness-saturation system.

```
win = New Window( "Color Wheel",
    Graph(
        Frame Size( 200, 200 ),
        For( hue = 0, hue < 360, hue += 30,
            y = 50 - 40 * Cos( hue * 2 * Pi() / 360 );
            x = 50 + 40 * Sin( hue * 2 * Pi() / 360 );
            Fill Color( HLS Color( hue / 360, 0.5, 1 ) );
            Oval( x - 10, y - 10, x + 10, y + 10, 1 );
        )
    )
);
```

Finally, Heat Color() returns the JMP color that corresponds to a value in any color theme that is supported by Cell Plot, Treemap, and so on. The syntax is:

```
Heat Color( n, <<"theme" )
```

The theme message is optional, and the default value is "Blue to Gray to Red". You can specify any color theme, including custom color themes. You can also create and use an anonymous color theme as shown in the following examples:

```
Heat Color( z, <<{"", {{1, 1, 0}, {0, 0, 1}} } );
Heat Color( z, <<{"", {blue, green, yellow} } );
```

Specify Transparency

In a graphics environment (like a Frame Box), use the Transparency function to set the level of transparency. The argument, alpha, can be any number between zero and one. The value 0 means clear and drawing has no effect. The value 1 means completely opaque and is the usual drawing mode. Intermediate values build semi-transparent color layers on top of what has already been drawn below it. The following example script illustrates transparency with rectangles.

```
win = New Window( "Transparency",
  Graph Box(
    Frame Size( 200, 200 ),
    Pen Color( "gray" );
    Fill Color( "gray" );
    Transparency( 0.25 );
    Rect( 0, 40, 60, 0, 1 );

    Pen Color( "red" );
    Fill Color( "red" );
    Transparency( 0.5 );
    Rect( 10, 60, 70, 10, 1 );

    Pen Color( "green" );
    Fill Color( "green" );
    Transparency( 0.75 );
    Rect( 50, 90, 90, 50, 1 );

    Pen Color( "blue" );
    Fill Color( "blue" );
    Transparency( 1 );
    Rect( 0, 80, 70, 70, 1 );
  )
);
```

Figure 12.20 Transparency and Rectangles

Add Fill Patterns

Fill Pattern() sets the pattern for filled areas.

Matrix Example

Specify a mask (a matrix of values between 0 and 1) or an image. Each value in the matrix creates a pixel.

```
win = New Window( "Example",
  Graph Box(
    Fill Pattern(
      [1 0.5 0 0, 0.5 0 0 1, 0 0 1 0.5, 0 1 0.5 0]
    );
    Polygon( [10 30 90], [88 22 44] );
  )
);
```

Named Pattern Example

Specify a name in quotation marks that corresponds to a pattern shown in Table 12.3.

```
win = New Window( "Named Pattern Example",
  Graph Box(
    Fill Pattern( "vertical light" );
    Polygon( [10 30 90], [88 22 44] );
  )
);
```

Graphic Example

Specify an image in quotation marks. If the image is not installed on the viewer's computer, question marks appear in the shape.

```
win = New Window( "Graphic Example",
  Graph Box(
    Fill Pattern( Open( "$SAMPLE_IMAGES/pi.gif", "gif" ) );
    Polygon( [10 30 90], [88 22 44] );
  )
);
```

Table 12.3 Fill Patterns

Pattern Name	Pattern
left slant light	

Table 12.3 Fill Patterns *(Continued)*

Pattern Name	Pattern
right slant light	
vertical light	
horizontal light	
grid light	
hatch light	
h wave light	
v wave light	
hollow circle	
left slant medium	
right slant medium	
vertical medium	
horizontal medium	
grid medium	
hatch medium	
h wave medium	
v wave medium	
filled circle	

Table 12.3 Fill Patterns *(Continued)*

Pattern Name	Pattern
left slant heavy	
right slant heavy	
vertical heavy	
horizontal heavy	
grid heavy	
hatch heavy	
h wave heavy	
v wave heavy	
diamond	
left slant heavy b	
right slant heavy b	
random	
square	
square offset	
wide	
tall	
checkerboard	

Table 12.3 Fill Patterns *(Continued)*

Pattern Name	Pattern
grid dots	
triangle up	
triangle down	
triangle left	
triangle right	
weave light	
weave heavy	
honeycomb	

Specify Line Types

You can also control Line Style() by number (0–4) or name (Solid, Dotted, Dashed, DashDot, DashDotDot). Figure 12.21 shows the numerical values for each line type.

```
linestyles = {"Solid", "Dotted", "Dashed", "DashDot", "DashDotDot"};
win = New Window( "Line Styles",
    Graph Box(
        Frame Size( 200, 200 ),
        X Scale( -1, 5 ),
        Y Scale( -1, 5 ),
        For( i = 0, i < 5, i++,
            Line Style( i );
            H Line( i );
            Text( {0, i + .1}, i );
            Text( {1, i + .1}, linestyles[i + 1] );
        )
    )
);
```

Figure 12.21 Line Styles

To control the thickness of lines, set a `Pen Size` and specify the line width in pixels. The default is 1 for single-pixel lines. For printing, think of `Pen Size` as a multiplier for the default line width, which varies according to your printing device.

```
win = New Window( "Pen Size",
    Graph Box(
        Pen Size( 2 ); // double-width lines
        Line( [10 30 90], [88 22 44] );
    )
);
```

Draw with Pixels

You can also draw using pixel coordinates. First, set the `Pixel Origin()` in terms of graph coordinates, and then use `Pixel Move To()` or `Pixel Line To()` in pixel coordinates relative to that origin. The main use for `Pixel()` functions is for drawing custom markers that do not vary with the size or scale of the graph. You can store a marker in a script and then call it within any graph. This example uses `Function()` to store pixel commands in a script with its own arguments, *x* and *y*.

```
ballotBox = Function( {x, y},
    Pixel Origin( x, y );
    Pixel Move To( -5, -5 );
    Pixel Line To( -5, 5 );
    Pixel Line To( 5, -5 );
    Pixel Line To( -5, -5 );
    Pixel Line To( 5, 5 );
    Pixel Line To( -5, 5 );
    Pixel Move To( 5, 5 );
    Pixel Line To( 5, -5 );
);
win = New Window( "Custom Markers",
    Graph Box(
```

```
        Frame Size( 200, 200 ),
        ballotBox( 10, 10 );
        ballotBox( 15, 90 );
        ballotBox( 20, 50 );
        ballotBox( 80, 50 );
        ballotBox( 60, 70 );
    )
);
```

Figure 12.22 Drawing Custom Markers

Interactive Graphs

Handle() and Mousetrap() are functions for making interactive graphs that respond to clicking and dragging. Handle() lets you parametrize a graph by adding a handle-marker that can be dragged around with the mouse, executing the graph's script at each new location. Mousetrap() is similar, but it takes its arguments from the coordinates of a click without dragging a handle. The main difference is that Handle() only catches mousedowns at the handle-marker's location, but Mousetrap() catches mousedowns at any location.

Another approach is to place buttons or slider controls outside the graph with Button Box(), Slider Box(), or Global Box().

Handle()

Handle() places a marker at the coordinates given by the initial values of the first two arguments and draws the graph using the initial values of the arguments. You can then click and drag the marker to move the handle to a new location. The first script is executed at each mousedown to update the graph dynamically, according to the new coordinates of the handle. The second script (optional, and not used here) is executed at each mouseup, similarly; see the example for

```
// Normal Density
mu = 0;
sigma = 1;
rsqrt2pi = 1 / Sqrt( 2 * Pi() );
win = New Window( "Normal Density",
  Graph Box(
    Frame Size( 500, 300 ),
    X Scale( -3, 3 ),
    Y Scale( 0, 1 ),
    Y Function( Normal Density( (x - mu) / sigma ) / sigma, x );
    Handle(
      mu,
      rsqrt2pi / sigma,
      mu = x;
      sigma = rsqrt2pi / y;
    );
    Text( {1, .7}, "mu ", mu, {1, .65}, "sigma ", sigma );
  )
);
```

Run demoPlotProb.jsl in the JMP Samples/Scripts folder to see Beta Density, Gamma Density, Weibull Density, and LogNormal Density graphs. The output for Normal Density is shown in Figure 12.23. Because we cannot show you the picture in motion, be sure to run this script yourself.

Figure 12.23 Normal Density Example for `Handle()`

To avoid errors, be sure to set the initial values of the handle's coordinates, as in the first line of this example.

If you want to use some function of a handle's coordinates, such as in the normal density example, you should adjust the arguments for `Handle()`. Otherwise, the handle marker would run away from the mouse.

```
Y Function( a * x ^ b );
Handle( a, b, a = 2 * x, b = y );
```

Suppose you drag the marker from its initial location to (3,4). The argument *a* is set to 6 and *b* to 4; the graph is redrawn as $Y = 6x^4$; and the handle is now drawn at (6,4), several units away from the mouse. To compensate, you would adjust the first argument to handle, for example.

```
Handle( a / 2, b, a = 2 * x; b=y );
```

To generalize, suppose you define the `Handle()` arguments as functions of the handle's (x, y) coordinates. For example, $a=f(x)$ and $b=g(y)$. If $f(x)=x$ and $g(y)=y$, then you would specify simply `a,b` as the first two arguments. If not, you would solve $a = f(x)$ for x and solve $b = g(y)$ for y to get the appropriate arguments.

You can use other functions to constrain `Handle()`. For example, here is an interactive graph to demonstrate power functions that uses `Round()` to prevent bad exponents and to keep the intercepts simple.

```
a = 3;
b = 2;
win = New Window( "Intercepts and Powers",
  Graph Box(
    Frame Size( 200, 200 ),
    X Scale( -10, 10 ),
    Y Scale( -10, 10 ),
    Y Function( Round( b ) + x ^ (Round( a )), x );
    Handle(
      a,
      b,
      a = x;
      b = y;
    );
    Text( {a, b}, " Move me" );
    Text( {-9, 9}, "y=", Round( b ), " + x^", Round( a ) );
  )
);
```

Figure 12.24 Intercepts and Powers for `Handle()`

`Handle()` and `For()` can be nested for complex graphs.

```
a = 5;
b = 5;
win = New Window( "Powers",
   Graph Box(
      Frame Size( 200, 200 ),
      X Scale( -10, 10 ),
      Y Scale( -10, 10 ),
      For( i = 0, i < 1.5, i += .2,
         Pen Color( 1 + 10 * i );
         Text Color( 1 + 10 * i );
         Y Function( i * x ^ Round( a ), x );
         Handle(
            a,
            b,
            a = x;
            b = y;
         );
         h = 9 - 10 * i;
         Text( {-9, h}, b, "*i*x^", Round( a ), ", i=", i );
      ) ) );
```

Figure 12.25 Nested Handle() and For()

You can use more than one handle in a graph:

```
amplitude = 1;
freq = 1;
phase = 0;
win = New Window( "Sine Wave",
    Graph Box(
        Frame Size( 500, 300 ),
        X Scale( -5, 5 ),
        Y Scale( -5, 5 ),
        Y Function( amplitude * Sine( x / freq + phase ), x );
        Handle( // first handle
            freq,
            amplitude,
            freq = x;
            amplitude = y;
        );
        Handle( phase, .5, phase = x ); // second handle
        Text(
            {3, 4},
            "amplitude: ",
            Round( amplitude, 4 ),
            {3, 3.5},
            "frequency: ",
            Round( freq, 4 ),
            {3, 3},
            "phase: ",
            Round( phase, 4 )
        );
    )
);
```

Figure 12.26 Two Handles

Mousetrap()

Mousetrap() takes arguments for a graph from the coordinates of a mouse click. The first
script is executed after each mousedown and the second script after each mouseup to update
the graph dynamically, according to the new coordinates of the handle. As with Handle, it is
important to set the initial values for the Mousetrap's coordinates. If you include both
Mousetrap() and Handle() in a graph, put the Handle() before the Mousetrap(). Then
Handle() has a chance to catch clicks before a Mousetrap() does.

This example uses both Mousetrap() and Handle() to draw a three-dimensional function
centered on the Mousetrap() coordinates, where the single contour line takes its value from a
Handle().

```
x0 = 0;
y0 = 0;
z0 = 0;
win = New Window( "Viewing a 3-D Function in Flatland",
  Graph Box(
    Frame Size( 300, 300 ),
    X Scale( -5, 5 ),
    Y Scale( -5, 5 ),
    Contour Function(
      Exp( -(x - x0) ^ 2 ) * Exp( -(y - y0) ^ 2 ) * (x - x0),
      x,
      y,
      z0 / 10
    );
```

```
             // get the z-cut values from a handle
             Handle( -4.5, z0, z0 = Round( y * 10 ) / 10 );
                   V Line( -4.5 );
             Text Size( 9 );
             Text(
                Counterclockwise,
                {-4.6, -4},
                "Drag to set the z-value for contour cut: z = " || Char( z0 / 10 )
             );
             Marker Size( 2 );
             Marker( 2, {x0, y0} );
             Mousetrap( // set the origin to the click-point
                x0 = x;
                y0 = y;
             );
             Text(
                {-4.25, -4.9},
                "Click any location to set the function's centerpoint."
             );
          )
       );
```

Note: Instead of setting x0 to x and y0 to y in the Mousetrap() expression, you might want to use Function({xx, yy}, Show(xx, yy); x0 = xx; y0 = yy);. xx and yy are formal parameters to the function. They come from arguments that the mousetrap passed when the user clicked on the graph. The Function({xx, yy}...) alternative makes the xx and yy names explicit, and makes it clear the you can name them anything, not just x, y.

Figure 12.27 Mousetrap() and Handle()

You might use `Mousetrap()` to collect points in a data table, such as for visually interpolating points in a graph. The following example illustrates a script that could be adapted and added to a data plot (such as a scatterplot from Fit Y by X) for that purpose. After you run the script and click in the graph, the point at which you click is recorded in the data table.

```
dt = New Table( "dat1" );
New Column( "xx", Numeric );
New Column( "yy", Numeric );
x = 0;
y = 0;
Add Point = Expr(
   dt << Add Rows( 1 );
   Row() = N Row();
   :xx = x;
   :yy = y;
);
win = New Window( "Add Points",
   Graph Box(
      Frame Size( 500, 300 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      For Each Row( Marker( {xx, yy} ) );
      Mousetrap( {}, Add Point );
   )
);
```

Notice that the first script argument is empty. At mousedown, nothing happens. The second script, add point, is executed at mouseup to add a data point. If you click, drag, and release, then the point that is added to your data set is the point where you let go of the mouse button; it is not the point where you pressed down the mouse button.

Drag Functions

There are five `Drag()` functions to perform similar functions to `Handle()` and `Mousetrap()` but with more than one point at a time. For *n* coordinates in matrices listed as the first two arguments:

- `Drag Marker()` draws *n* markers.

- `Drag Line()` draws a connected line with *n* vertices and *n* – 1 segments.

- `Drag Rect()` draw a filled rectangle using the first two coordinates, ignoring any further coordinates.

- `Drag Polygon()` draws a filled polygon with *n* vertices.

- Drag Text() draws a text item at the coordinates. Or if there is a list of text items, it draws the *i*th list item at the *i*th (*x,y*) coordinate. If there are fewer list items than coordinate pairs, the last item is repeated for remaining points.

The syntax for these commands:

```
Drag Marker( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Line( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Rect( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Polygon( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Text( xMatrix, yMatrix, "text", dragScript, mouseupScript );
```

They all must have L-value arguments for the coordinates. The coordinates are literal matrices or names of matrix values that are modified if you click a vertex and drag it to a new position. The script arguments are optional and behave the same as with Handle(). However, there is no *x* nor *y* that is modified as in Handle().

Drag() functions are ways to display data that the user can adjust and then capture the adjusted values. Consider the earlier script to draw the JMP man. Drag Polygon() makes it possible to draw an editable JMP man; using a matching Drag Marker() statement makes the vertices more visible. And, similar to the Mousetrap() example, you can save the new coordinates to a data table. Notice how : and :: operators avoid ambiguity among matrices and data table columns with the same names.

You could just as easily put *storepoints* in the fourth argument of Drag Polygon() or Drag Marker(), but that would create a data table after each drag. You probably just want a single data table when you are finished. Regardless, the values in *gCoordX* and *gCoordY* update with each drag.

```
::i = 1;
storepoints = Expr(
  mydt = New Table( "My coordinates" || Char( i ) );
  i++;
  New Column( "GCoordX", Numeric );
  New Column( "GCoordY", Numeric );
  mydt << Add Rows( N Row( GcoordX ) );
  :GCoordX << Values( ::GcoordX );
  :GCoordY << Values( ::GcoordY );
);

::GcoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75,
  12.5, 6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375,
  37.5, 38.75, 40.625, 42.5, 43.125, 42.5, 41.25, 38.75, 43.75, 50, 56.25,
  60.625, 63.75, 65.625, 62.5, 56.25, 50, 45, 37.5, 32.5, 28.75, 27.5, 26.25,
  25.625, 25];
::GcoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41,
  40, 39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40,
  38, 36, 37, 37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
```
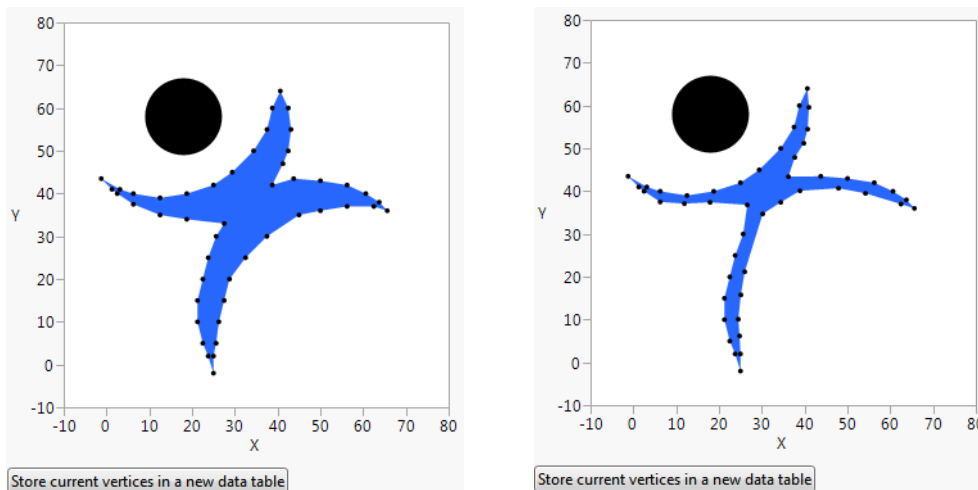
```
win = New Window( "Redraw the JMP Man!",
   V List Box(
      Graph Box(
         Frame Size( 300, 300 ),
         X Scale( -10, 80 ),
         Y Scale( -10, 80 ),
         Fill Color( "blue" );
         Drag Polygon( GcoordX, GCoordY );
         Pen Color( "gray" );
         Drag Marker( GcoordX, GCoordY );
         Fill Color( {0, 0, 0} );
         Circle( {18, 58}, 9, "FILL" );
      ),
      Button Box( "Store current vertices in a new data table", storepoints )
   )
);
```

Suppose you want the body of the JMP man to be smaller. Here is how he looks before and after some judicious vertex-dragging. Clicking the button after re-shaping the JMP Man executes the *storepoints* script to save his new, slender figure in a data table of coordinates.

Figure 12.28 Redraw the JMP Man

This example uses two functions discussed under "Constructors for New Windows" on page 542 in the "Display Trees" chapter:

- `Button Box()`, which creates controls outside the graph.
- `V List Box()`, which combined the graph box and the button box in the same graph window.

Troubleshoot Interactive Graphs

If your interactive graphs do not work as expected, make sure that you supply initial values for the Handle() or Mousetrap() coordinates (and other globals as needed). Also make sure that the values make sense for the graph.

Create a Color Picker

Let users select a color to apply to graphs by creating a color picker. Users can select a predefined color or create their own color. You can also specify a default color in your script.

- Use the named or numbered colors shown in "Colors and Markers" on page 451 in the "Data Tables" chapter.

  or

- Specify values between 0 and 1 for each of the red, blue, and green components in RGB Color().

Examples of Creating Color Pickers

The following example does not name the window, so the default title is used. Because the script does not specify a color, Black is selected by default. If the user selects a color in the color picker window, then the lines are drawn in the selected color.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Run Script( "Bivariate" );
Wait( 0 );
pickColor = Pick Color();
Report( biv )[AxisBox( 1 )] <<
Add Ref Line( 105, "Dashed", pickColor, "Mean" );
```

The following example creates a color picker with the window title "Select a Line Color". BlueGreen is preselected.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Run Script( "Bivariate" );
Wait( 0 );
pickColor = Pick Color( "Select a Line Color", "BlueGreen" );
Report( biv )[AxisBox( 1 )] <<
Add Ref Line( 105, "Dashed", pickColor, "Mean" );
```

The following example creates a color picker with the window title "Select a Background Color". BlueGreen is preselected.

```
picked color = Pick Color( "Select a Background Color", "BlueGreen" );
New Window( "Example",
```

```
        gb = Graph Box(
              Frame Size( 300, 300 ),
              Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
              Line( [10 30 70], [88 22 44] );
        )
);
gb[FrameBox(1)] << Background Color ( picked color );
```

The following example specifies the RGB values for red, green, and blue to create pink. If you do not know the JMP color name or number, you might consider this option.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Run Script( "Bivariate" );
Wait( 0 );
pickColor = Pick Color( "RGB", RGBCOLOR( 1/*red*/, 0/*green*/, 1/*blue*/ ) );
Report( biv )[AxisBox( 1 )] <<
Add Ref Line( 105, "Dashed", pickColor, "Mean" );
```

Clip Display Segs in Graphics

Display segs (such as reference lines, grid lines, and contours) can be clipped to conform to the natural boundaries of the underlying data. This gives you a better view of the boundaries. For example, Figure 12.29 shows contours that are clipped to the boundaries of a map of the United States.

Figure 12.29 Unclipped (Left) and Clipped (Right) Boundaries

Send the `Clip Shape` message to the Contour Seg object to clip the display segs.

```
Open( "$SAMPLE_DATA/Cities.jmp" );
gb = Graph Builder(
    Size( 653, 396 ),
    Show Control Panel( 0 ),
    Variables( X( :Longitude ), Y( :Latitude ) ),
    Elements( Contour( X, Y, Legend( 2 ) ) ),
```

```
      SendToReport(
         Dispatch(
            {},
            "Graph Builder",
            FrameBox,
            {Background Map( Boundaries( "US States" ) ), Grid Line Order( 2 ),
      Reference Line Order( 3 )}
         )
      )
   );
   cs = (gb << Report)[FrameBox( 1 )] << Find Seg( Contour Seg( 1 ) );
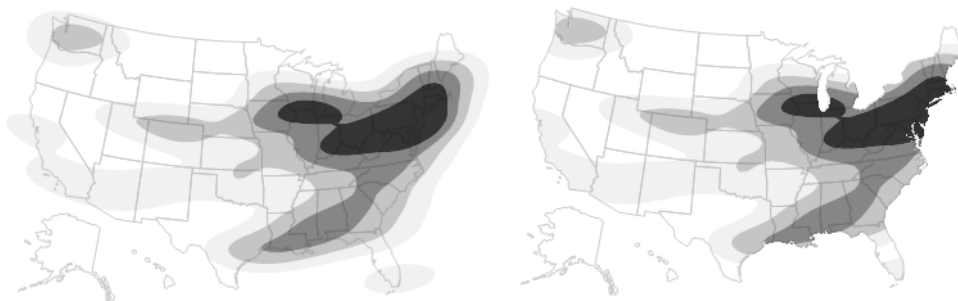
   Wait( 2 );

   cs << Clip Shape( Boundaries( "US States" ) );
```

Tip: Clip Shape(Empty()) removes the clipping path.

Here's an example of defining the clip shape path with a matrix:

```
   Open( "$SAMPLE_DATA/Big Class.jmp" );
   gb = Graph Builder(
      Show Control Panel( 0 ),
      Variables( X( :height ), Y( :weight ), Overlay( :sex ) ),
      Elements( Contour( X, Y, Legend( 3 ) ) )
   );
   r = ( gb << Report );
   cs = r[FrameBox( 1 )] << Find Seg( Contour Seg( 1 ) );
   cs << Clip Shape( Path( [60 80 1, 50 140 2, 65 120 2, 70 65 -2] ) );
```

Figure 12.30 Unclipped (Left) and Clipped (Right) Contours

The matrix [60 80 1, 50 140 2, 65 120 2, 70 65 -2] indicates the following:

1. Start at 60, 80.

2. Draw a line to 50, 140.

3. Draw a line to 65, 120.

4. Draw a line to 70, 65.

5. Connect back to the first point.

Create Background Maps

Background maps can be scripted in JSL. You can write a script that creates a graph and then turns on the background map in the script.

There are two types of background maps: `Images()` and `Boundaries()`. Each of these takes a parameter, which is the name of the map to use. The name is one of the maps listed in the window.

- For `Images()`, the choices are Simple Earth, Detailed Earth, NASA, Street Map Service, and Web Map Service. If you use Web Map Service, then there are two additional parameters: the WMS URL and the layer supported by the WMS server.

- For `Boundaries()`, the choices vary since boundaries can be user-defined. A typical choice is World.

The following example uses the Simple Earth as an image and World as a boundary:

```
dt = Open( "$SAMPLE_DATA/Air Traffic.jmp" );
dt << Graph Builder(
   Size( 1101, 603 ),
   Show Control Panel( 0 ),
   Variables( X( :Longitude ), Y( :Latitude ) ),
   Elements( Points( X, Y, Legend( 8 ) ) ),
   SendToReport(
      Dispatch(
         {},
         "Graph Builder",
         Frame Box,
         {Background Map( Images( "Simple Earth" ), Boundaries( "World" ) ),
         Grid Line Order( 3 ), Reference Line Order( 4 )}
      )
   )
);
```

Figure 12.31 shows an excerpt from the map:

Figure 12.31 JSL Scripting Example

To change the script to use a WMS server, the command would look like this:

```
Background Map ( Images ( "Web Map Service",
    "http://sedac.ciesin.columbia.edu/geoserver/wms",
    "gpw-v3:gpw-v3-population-density_2000" ), Boundaries ( "US States" ) )
```

To see which layers are available on a WMS server, install the WMS Explorer Add-In. Download the add-in from the JMP File Exchange at https://community.jmp.com/docs/DOC-6095. Note that some WMS servers are not reliable. If the server is down, or if the user does not have an Internet connection, the WMS map does not appear.

Tips:

- To see the syntax for the JSL script, add a background map through the user interface. Then from the red triangle menu, select **Save Script > To Script Window** to see the script that is generated.

- To specify image and boundary names in the JSL script, use the names shown in the Set Background Map window. (Right-click a map and select **Graph > Background Map**).

- Look at sample data to find examples of background map scripts. Select **Help > Sample Data Library** and open Napoleons March.jmp, Pollutants Map.jmp, or San Francisco Crime.jmp. Right-click the map table script and select **Edit** to see the Background Map() function.

Graph Elements

You can use the following commands inside Graph Box() statements. This chapter focuses on the JSL that is specific to graphing, but you can also use general script commands such as For and While.

Note: New Window() expects window and display box commands, and platforms expect
platform commands. Therefore, the general script commands such as For and While might
not work inside platforms or New Window(), or you might get unexpected results.

Plot Functions

A Y Function() function is used to draw smooth functions. The first argument is the
expression to be plotted. The second argument is the name of the X variable in the expression.

```
win = New Window( "Sine Function",
    Graph Box(
        Frame Size( 200, 100 ),
        X Scale( -10, 10 ),
        Y Scale( -1, 1 ),
        X Name( "x" ),
        Y Name( "Sine(x)" ),
        Y Function( Sine( x ), x ) ) );
```

Figure 12.32 Sine Wave

You can use For to overlap several sine waves:

```
win = New Window( "Overlapping Sine Waves",
    Graph Box(
        Frame Size( 200, 100 ),
        X Scale( -10, 10 ),
        Y Scale( -1, 1 ),
        For( i = 1, i <= 4, i += .1,
            Y Function( Sine( x / i ), x )
        )
    )
);
```

Figure 12.33 Overlapping Sine Waves

Similarly, X Function() is for drawing a graph where the symbol is varied on the Y variable.

```
win = New Window( "Overlapping Sine Waves",
   Graph Box(
      Frame Size( 100, 200 ),
      X Scale( -1, 1 ),
      Y Scale( -10, 10 ),
      For( i = 1, i <= 4, i += .2,
         X Function( Sine( y / i ), y )
      )
   )
);
```

Figure 12.34 Overlapping Sine Waves along the X-Axis

An XY Function() draws a smooth curve using a pair of formulas (parametric equations) that depend on a third variable. The third variable's value is incremented from a minimum value to a maximum value to generate the X-Y pairs.

```
win = New Window( "Spiral",
   Graph Box(
      Pen Color( "red" ); // red line color
      xCenter = 50; // location of the center of the curve on the x axis
      yCenter = 50; // location of the center of the curve on the y axis
      minAngle = 0;
      maxAngle = Pi() * 2 * 20;
      XY Function(
```

```
            xCenter + ((ta / 3) * Cos( ta )),
            yCenter + ((ta / 3) * Sin( ta )),
            ta,
            Min( minAngle ),
            Max( maxAngle ),
            Inc( Pi() / 100 )
        );
    )
);
```

In this example, `Sin()` and `Cos()` use `ta` as an argument (rotates) *and* as a factor (expands). (`Sin()` and `Cos()` use radians, not degrees.)

Figure 12.35 Spiral Parametric Plot

`Contour Function()` is an analogous way to represent a three-dimensional function in a two-dimensional space. The final argument specifies the value(s) for the contour line(s). The argument can be a value, an indexed range of values using `::`, or a matrix of values.

```
win = New Window( "Bird's eye view of the egg carton function",
    Graph Box(
        Frame Size( 300, 300 ),
        X Scale( -10, 10 ),
        Y Scale( -10, 10 ),
        Pen Color( "black" );
        Pen Size( 2 );
        Contour Function( Sine( y ) + Cosine( x ), x, y, (0 :: 20) / 5 );
        Pen Color( "red" );
        Pen Size( 1 );
        Contour Function( Sine( y ) + Cosine( x ), x, y, (-20 :: 0) / 5 );
    )
);
```

Figure 12.36 Egg Carton Function

`Normal Contour()` draws normal probability contours for *k* populations and two variables. The first argument is a scalar probability or a matrix of probability values for the contours. Subsequent arguments are matrices to specify means, standard deviations, and correlations. The mean and standard deviation matrices have dimension *k* × 2. The correlation matrix should be *k* × 1, where the first row pertains to the first contour, the second row to the second contour, and so on. The first column is for *x* and the second column for *y*. Consider the following example:

```
Normal Contour(
    [ prob1,
      prob2,
      prob3, ...],
    [ xmean1 ymean1,
      xmean2 ymean2,
      xmean3 ymean3, ...],
    [ xsd1 ysd1,
      xsd2 ysd2,
      xsd3 ysd3, ...],
    [ xycorr1,
      xycorr2,
      xycorr3, ...]);
```

The following script draws contours at probabilities 0.1, 0.5, 0.7, and 0.99 for two populations and two variables. The first population has *x* mean 0 and *y* mean 1, with standard deviation 0.3 along the *x* axis and 0.6 along the *y*-axis, and with correlation 0.5. The second has *x* mean 4 and *y* mean 6, with standard deviation 0.8 along the *x* axis and 0.4 along the *y*-axis, and with correlation 0.9.

```
win = New Window( "Normal Contours",
    Graph Box(
```

```
        X Scale( -5, 10 ),
        Y Scale( -5, 10 ),
        Normal Contour( [.1, .5, .7, .99], [0 1, 4 6], [.3 .6, .8 .4], [.5, .9] )
    )
);
```

Figure 12.37 Normal Contour Function

`Normal Contour()` is a general way to accomplish effects such as Bivariate's density ellipses. The Bivariate script in the Football.jmp sample data creates an example.

Gradient Function

`Gradient Function()` fills a set of rectangles on a grid according to a color determined by the expression value as it crosses a range corresponding to a range of colors.

> Gradient Function(*expression*, *xname*, *yname*, [*zlow*, *zhigh*], ZColor(
>   [*colorLow*, *colorHigh*]), <XGrid(*min*, *max*, *incr*)>, <YGrid(*min*, *max*,
>   *incr*)>, <Transparency(t)>;

Here are explanations of the elements in the `Gradient Function()` syntax:

```
GradientFunction(
    /* the expression to be contoured, which is a function in terms of
    the two variables that follow expression*/
    Expression,
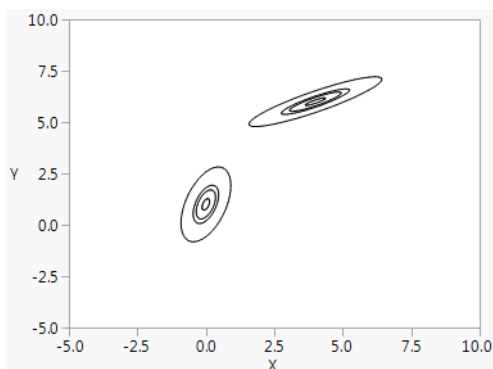
    // the two variable names used in the expression
    xname, yname,

    // the low and high expression values the gradient is scaled between
    [zlow, zhigh],

    // the colors that correspond to the low value and the high value
    ZColor([colorLow, colorHigh])
```

```
// optional specification for the grid of values
<XGrid(min, max, incr),>
<YGrid(min, max, incr)> );

/* the value 0 means clear. The value 1 means completely opaque and
is the usual drawing mode. */
<Transparency>
```

The ZColor() values must be numeric codes rather than names. You can use the color menu indices (such as 0=black, 1=grey, 2=white, 3=red, 4=green, and 5=blue) found in "Specify Colors" on page 659.

The following example uses Gradient Function() to create an animated graph.

```
phase = 0.7;
win = New Window( "Gradient Function",
  a = Graph(
    Frame Size( 400, 400 ),
    X Scale( -5, 5 ),
    Y Scale( -5, 5 ),
    Gradient Function(
      phase * Sine( x ) * Sine( y ) + (1 - phase) * Cosine( x ) * Cosine( y
  ),
      x,
      y,
      [-1 1],
      zcolor( [0, 2] )
    )
  )
);
b = a[FrameBox( 1 )];
For( i = 1, i <= 5, i++,
  For( phase = 0, phase < 1, phase += 0.05,
    b << Reshow;
    Wait( 0.01 );
  );
  For( phase = 1, phase > 0, phase -= 0.05,
    b << Reshow;
    Wait( 0.01 );
  );
);
```

Figure 12.38 Gradient Function

Get the Properties of a Graphics Frame

Several functions are useful for getting properties of an existing graphics frame:

H Size   Returns the horizontal size of the graphics frame in pixels.

V Size   Returns the vertical size of the graphics frame in pixels.

X Origin   Returns the x-value for the left edge of the graphics frame.

X Range   Returns the distance from the left to right edges of a display box.

Y Origin   Returns the y-value for the bottom edge of the graphics frame.

Y Range   Returns the distance from the bottom to top edges of a display box.

In the following expressions, the first line calculates the right edge. The second line calculates the top edge.

```
Oval(
    ...,
    rightEdge = X Origin() + X Range();
    topEdge = Y Origin() + Y Range();
);
```

Add a Legend

To add a legend to a graph, send the Row Legend message to the frame box. In the Row Legend message, specify which column you want to base the legend on and whether the legend affects colors and markers.

For example, using Big Class.jmp, submit the following JSL to turn on a legend based on the
age column. Set both colors and markers by values in the age column.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :height ), x( :weight ) );
rbiv = biv << Report;
framebox = rbiv[Frame Box( 1 )];
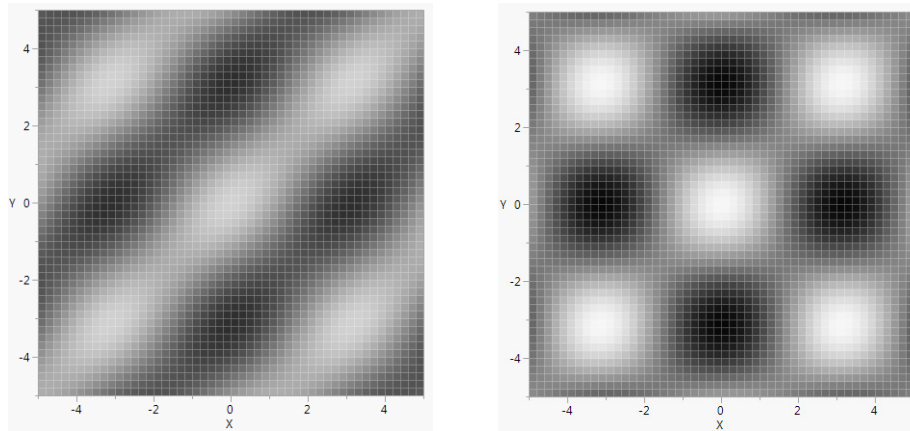
/* make a legend with the "age" title
display colors and markers in the legend */
framebox << Row Legend( "age", Color( 1 ), Marker( 1 ) );
```

The Color() and Marker() arguments are optional. Colors are on and markers are off by
default.

To use a continuous scale if your variable is nominal or ordinal, you can use a continuous
color theme with the Continuous Scale(1).

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :weight ), x( :height ) );
rbiv = biv << Report;
framebox = rbiv[Frame Box( 1 )];
framebox << Row Legend( "age", Color( 1 ), Continuous Scale( 1 ) );
```

To interactively add a legend, right-click the graph, select **Row Legend**, and change the
column settings.

Hover Labels

When you hover over a visual element in a graph, usually you see a basic description of the
data point on a hover label.

You can customize what appears on the hover label using the following methods:

- "Add Rich Text to Hover Labels Using Textlets"
- "Add Graphs or Images to Hover Labels Using Graphlets"
- "Update Names and Values in Hover Labels Using Gridlets"
- "Create Drill-Down Graphs"
- "Work with the Hover Label Execution Context"

Add Rich Text to Hover Labels Using Textlets

A *textlet* adds richly formatted text to the hover label. You might use a textlet to describe data in an explanatory paragraph or to emphasize important information by making it bold or applying a color.

The textlet in Figure 12.39 shows text that is formatted in a gray box with red text.

Figure 12.39 Textlet Example

The following script creates the textlet shown in Figure 12.39.

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA/Diabetes.jmp" );
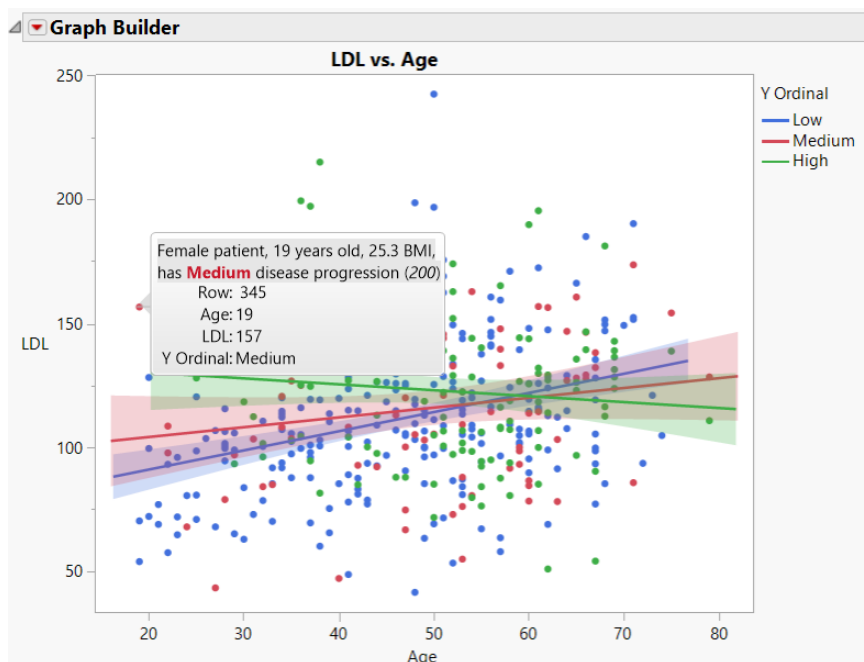
// create the baseline visualization
gb = dt << Graph Builder(
    Size( 534, 456 ),
    Show Control Panel( 0 ),
    Variables( X( :Age ), Y( :LDL ), Overlay( :Y Ordinal ) ),
    Elements( Points( X, Y, Legend( 8 ) ), Line Of Fit( X, Y, Legend( 10 ) ) )
);

// get a handle to the graph frame box
rpt = gb << Report;
```

```
        frame = rpt[Framebox( 1 )];

        /* configure the textlet -- a hover label extension that adds
           a rich text paragraph to the hover label. */
        frame << Set Textlet(
            /* "Setup" is a JSL fragment that uses Hover Label Execution Contect
               variables to define content that will be embedded in the
               rich text paragraph.*/
          Setup(
            // map numerical values to text
            local:gender = If( Column( local:_dataTable, "Gender" )[local:_firstRow]
              == "1",
              "Male",
              "Female"
            );
            // map a categorical value to text color
            local:color = Match( local:_Y Ordinal,
              "Low", "Medium Dark Blue",
              "Medium", "Medium Dark Red",
              "High", "Medium Dark Green"
            );
            // capture values from other columns for the current row
            local:progression = Column( local:_dataTable, "Y" )[local:_firstRow];
            local:BMI = Column( local:_dataTable, "BMI" )[local:_firstRow];
          ),
          /* markup defines the rich text template. It supports HTML-like markup such
          as bold, italic and other font tags. Variable references between braces
          ({}) are substituted with their actual value. */
          Markup(
            "{local:gender} patient, {local:_age} years old, {local:BMI} BMI,
            has <font color='{local:color}'><b>{local:_Y Ordinal}</b></font>
            disease progression (<i>{local:progression}</i>)"
          )
        );
        // pin a hover label to display a textlet-created rich text paragraph.
        frame << Add Pin Annotation(
          Seg( Marker Seg( 2 ) ),
          Index( 64 ),
          Index Row( 344 ),
          UniqueID( 568154464 ),
          FoundPt( {96, 222} ),
          Origin( {18.68125, 156.93706122449} ),
          Tag Line( 1 )
        );
```

Texlet Commands

See the example in "Add Rich Text to Hover Labels Using Textlets" on page 692 for more
information about writing texlet expressions.

Setup

Setup() defines variables used in the rich text that is displayed on the hover label.

```
...
frame << Set Textlet(
    /* "Setup" is a JSL fragment that uses Hover Label Execution Contect
      variables to define content that will be embedded in the
      rich text paragraph.*/
   Setup(
      // map numerical values to text
      local:gender = If( Column( local:_dataTable, "Gender" )[local:_firstRow]
         == "1",
         "Male",
         "Female"
      );
      // map a categorical value to text color
      local:color = Match( local:_Y Ordinal,
         "Low", "Medium Dark Blue",
         "Medium", "Medium Dark Red",
         "High", "Medium Dark Green"
      );
      // capture values from other columns for the current row
      local:progression = Column( local:_dataTable, "Y" )[local:_firstRow];
      local:BMI = Column( local:_dataTable, "BMI" )[local:_firstRow];
   )
...
```

In the interface, Setup is called "JSL Variables" on the Textlets Markup tab.

Markup

Markup() specifies a rich text paragraph template based on a subset of HTML tags. Can
contain JSL variable references enclosed in delimiters. The default delimiters are braces ({}).

```
...
Markup(
    "{local:gender} patient, {local:_age} years old, {local:BMI} BMI,
    has <font color='{local:color}'><b>{local:_Y Ordinal}</b></font>
    disease progression (<i>{local:progression}</i>)"
   )
```

In the interface, Markup is called "HTML Markup" on the Textlets Markup tab.

See *Using JMP* for more information about the supported font formatting tags.

Delimiters

`Delimiters()` specifies the delimiters that indicate variables for replacement in the Markup area. For example, you might use square bracket delimiters for JMP variables if the script contains C++ code.

```
Delimiters( "[]" )
```

In the interface, Delimiters is on the Textlets Other tab.

Width

`Width()` enables you to specify a preferred width in pixels for the hover label.

```
Width( 300 )
```

In the interface, Width is on the Textlets Other tab.

Add Graphs or Images to Hover Labels Using Graphlets

You can add a thumbnail of a related graph or an image to a hover label using a *graphlet*. These graphs and images can have actions associated with them that are triggered by a mouse click. For example, you can click a graph to launch it, or click an image to open a related link.

Example of a Graphlet in Graph Builder

Click the JMP man graph shown in Figure 12.40 to open it in Graph Builder and customize the graph. See *Using JMP* for details about opening a graph in a platform, customizing the graph, and then saving the updated graph in the original graph.

Figure 12.40 Graphlet Example

The following script creates the graph and hover label that are shown in Figure 12.40.

```
Names Default To Here( 1 );
// open the data table
dt = Open( "$SAMPLE_DATA/JMP Man Dozen.jmp" );
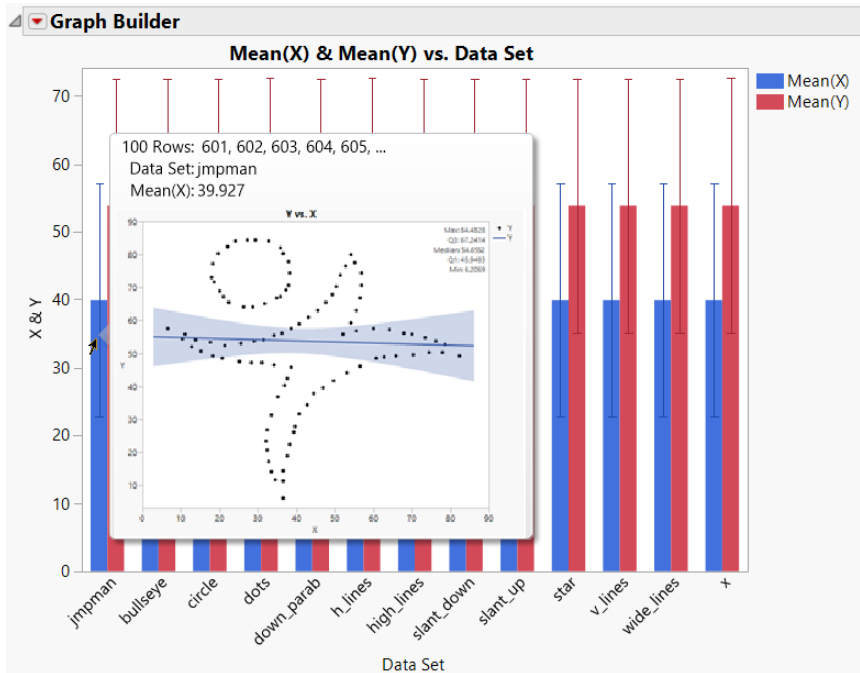
// launch the baseline visualization
gb = Graph Builder(
	Size( 529, 466 ),
	Show Control Panel( 0 ),
	Variables( X( :Data Set ), Y( :X ), Y( :Y, Position( 1 ) ) ),
	Elements(
		Bar(
			X,
			Y( 1 ),
			Y( 2 ),
			Legend( 12 ),
			Error Interval( "Standard Deviation" )
		)
	)
);

// get a reference to the visualization frame
```

```
    frame = (gb << Report)[FrameBox(1)];

    /* use the "Set Graphlet" command to define the visualization
    that will be rendered inside the hover label using the
    "Picture" attribute */
    frame << Set Graphlet(
        /* The value of the "Picture" attribute is a JSL fragment
          that is evaluated to generate a thumbnail. The thumbnail is
          displayed inside the hover label. In this case, the hover
          label shows a scatterplot with a caption box. */
       Picture(
            Graph Builder(
                Show Control Panel( 0 ),
                Variables( X( :X ), Y( :Y ) ),
                Elements(
                    Points( X, Y, Legend( 1 ) ),
                    Line Of Fit( X, Y, Legend( 3 ) ),
                    Caption Box( X, Y, Legend( 4 ),
                     Summary Statistic( "5 Number Summary" ) ) )
                )
            )
        /* Because the optional "Click" attribute is not defined, the
            "Picture" attribute script is re-used to launch the associated
            platform when the user clicks the graphlet thumbnail. */
        )
    );
```

Example of a Graphlet in Principal Components

Create a graphlet in the Principal Components platform. When the user clicks the hover label, the Principal Components report window appears. The user can then explore and filter the data.

Figure 12.41 Prinicipal Components Graphlet

The following script creates the graphlet shown in Figure 12.41:

```
dt = Open( "$SAMPLE_DATA/Iris.jmp" );
gb = dt << Graph Builder(
   Size( 531, 451 ),
   Show Control Panel( 0 ),
   Variables( X( :Species ), Y( :Sepal width ) ),
   Elements( Treemap( X, Y, Legend( 4 ) ) ),
   SendToReport(
      Dispatch(
         {},
         "Graph Builder",
         FrameBox,
         {Set Graphlet(
            Picture(
               Principal Components(
                  Y( :Sepal length, :Sepal width, :Petal length, :Petal width
),
                  Z( :Species ),
                  Estimation Method( "Default" ),
                  "on Correlations"
               )
            )
         )}
```

```
        )
      )
    );
```

Graphlet Commands

Picture

A required script that adds a thumbnail of a graph or an image to the hover label.

`Picture()` is always executed in a Hover Label Execution Context derived from the current visual element. The first `Picture Box` (outside of a local data filter) found will be used to generate the displayed image.

If `Picture()` is a call to a platform, a local data filter is included.

```
...
Picture(
        Graph Builder(
            Show Control Panel( 0 ),
            Variables( X( :X ), Y( :Y ) ),
            Elements(
                Points( X, Y, Legend( 1 ) ),
                Line Of Fit( X, Y, Legend( 3 ) ),
                Caption Box( X, Y, Legend( 4 ),
                 Summary Statistic( "5 Number Summary" ) )
            )
        )...
```

See "Add Graphs or Images to Hover Labels Using Graphlets" on page 695 for a complete example.

Click

An optional script for user-defined actions to be executed when the user clicks the associated image.

• Clicking runs the main graph's platform (for example, Graph Builder), which launches in its own window.

• If Click is not provided, the script provided in `Picture` is evaluated in a Hover Label Execution Context scope.

Skip Filters

An optional script. By default, any code executed in the context of a hover label customization (such as a graphlet) executes in a context defined in the hovered visual element (for example, a bar in a bar chart). This context includes a local data filter based on the current visual role assignment, and it is usually what you want. However, in some rare cases, you don't want the filter, or maybe part of the filter, so that the next graph generated by the graphlet can include more of the original data.

- If the Skip Filters field evaluates to a non-0 value (usually a positive integer such as 1) the selected Skip Filters Columns will be removed from the filter that will be applied to the new graphlet.

- Otherwise (for example, if it evaluates to 0), the selected columns are ignored and the filter is created as usual.

- If the Skip Filters Flag field is empty and no columns are selected in Skip Filters Columns (the default), then no columns are skipped when creating filters for the new window. Note that Skip Filters Flag is a JSL expression, which can be as complex as needed.

- If Skip Filters Flag field is not empty, but Skip Filter Columns is empty (no columns are selected), then all columns are filtered; the graphlet or new visualization includes all the data used by the original visualization. See "Example of Defining the Context of a Scatterplot Matrix and Ellipse" on page 718.

Suppose that you are looking at crime over time in the San Francisco Crime.jmp sample data table. You use a line chart where X = Date, an overlay of Police District, and no Y assignment. Lines represent counts over time per district. Now you want to use a graphlet to look at crime over time inside each district, broken down by category. Create this new graph and select **Save > to Clipboard** and then **Paste Graphlet** on the original graph to create the graphlet. However, the hovered graph shown in Figure 12.42 is not what you expect.

Figure 12.42 Initial Filtered Data

JMP created a filter based on both roles, X and the overlay. This is easier to see if you click on the graphlet to launch the new graph, and then expand the local data filter (on the left, collapsed by default). The filter on a single day causes the line chart to collapse into what looks like a marker plot as shown in Figure 12.43.

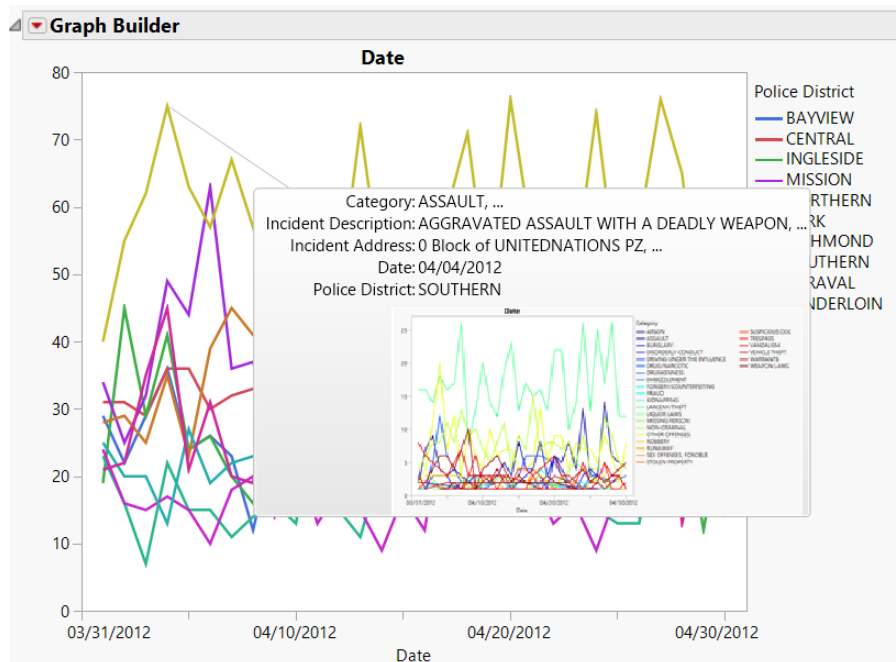**Figure 12.43** Local Data Filter with a Line Chart that Looks like a Marker Plot

You want the overlay value to restrict what you see in the new graph. However, you don't want the date value to do so. You still want to filter by the "SOUTHERN" Police District, but you do not want to filter on "04/14/2012" Date.

The solution is to use the Skip Filters Flag to remove the filter on Date from your original graph. On the Hover Label Editor, the script looks like this:

Figure 12.44 Example of Skip Filters Flag script in the Hover Label Editor

After the filter is removed, the end result is what you expect as shown in Figure 12.45.

Figure 12.45 Line Chart where Skip Filters Flag = 1

The following script creates the graph shown in Figure 12.45.

```
Names Default To Here( 1 );
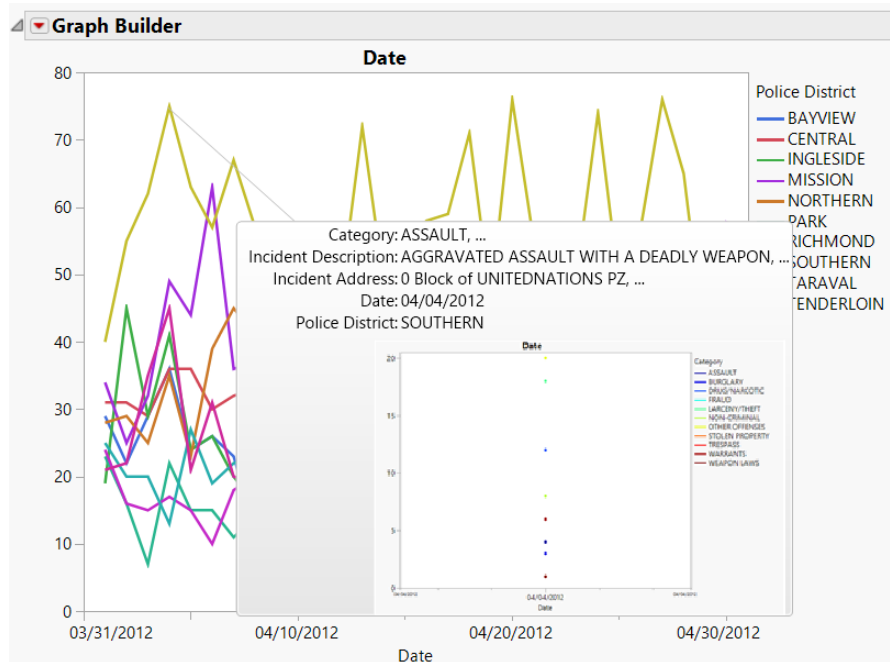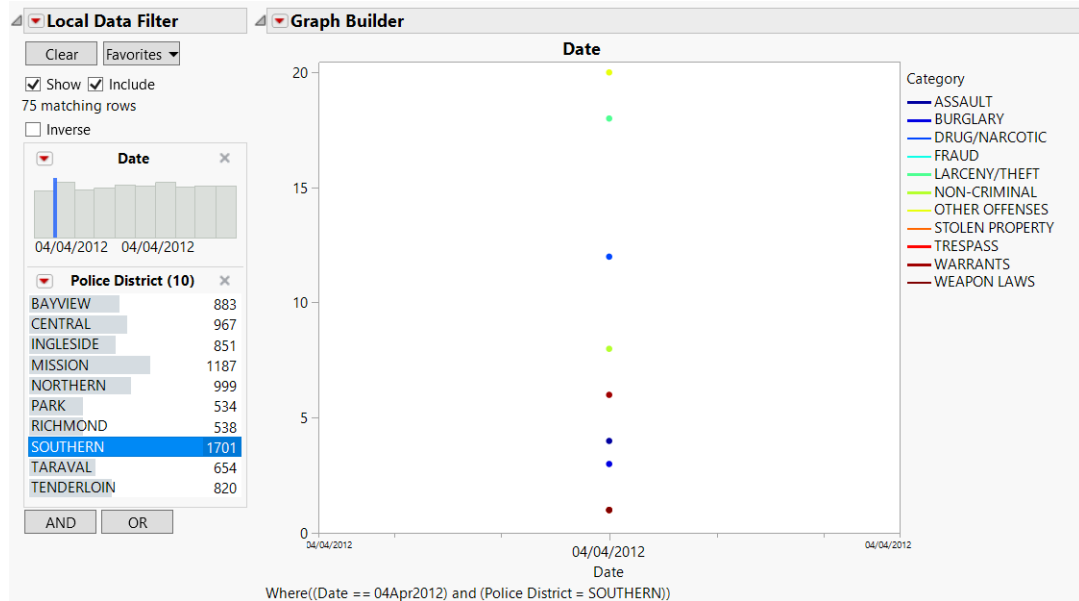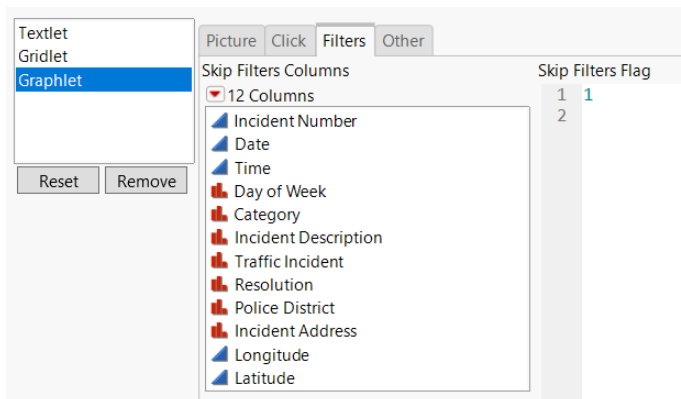
dt = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
gb = dt << Graph Builder(
	Size( 528, 456 ),
	Show Control Panel( 0 ),
	Variables( X( :Date ), Overlay( :Police District ) ),
	Elements( Line( X, Legend( 4 ) ) ),
	SendToReport(
		Dispatch(
			{},
			"Graph Builder",
			FrameBox,
			{Set Graphlet(
				Picture(
					Graph Builder(
						Size( 528, 456 ),
						Show Control Panel( 0 ),
						Variables( X( :Date ), Overlay( :Category ) ),
						Elements( Line( X, Legend( 1 ) ), Smoother( X, Legend( 2 ) ) )
					)
```

```
            ),
            Skip Filters( 1, {:Date} )
        )}
    )
  )
);
```

The Skip Filters Flag field holds an expression that is evaluated in the hovered visual element context. If expression evaluates to a truthful value (usually a positive integer such as 1), the selected "Skip Filters Columns" will be removed from the filter that will be applied to the new graphlet. Otherwise (for example, if it evaluates to zero), the selected columns are ignored and the filter is created as usual.

Title

Title is an expression that evaluates to a string. The result is the title that appears on the title bar of the window that contains the graph. Variables defined in the Hover Label Evaluation Context can be used in the Title expression.

For example, the following expression indicates the baseline visualization type and how many drilling levels have been used so far:

```
Title("Graphlet for " || local:_displaySegName ||
        ", depth is " || Char( local:_drillDepth ))
```

If the Title expression is not specified, the new window is named "Graphlet".

Note: In the JMP interface, Title is on the Hover Label Editor Other tab of a graphlet.

Reapply

Reapply is an expression that indicates whether the graphlet definition should be reapplied to new graphs or platforms created by clicking on a graphlet image. This is the core mechanism behind the ability to drill down using graphlets.

```
Reapply( local:_underlyingRows > 1 )
```

Note: In the JMP interface, Reapply is on the Hover Label Editor "Other" tab of a graphlet.

If Reapply is empty (the default setting), the graphlet definition is not reapplied. New windows launched by clicking on this graphlet doesn't show graphlets on its hover labels.

Update Names and Values in Hover Labels Using Gridlets

A *gridlet* supports the full customization of the hover label name and value pairs that reflect the underlying visual data. Data can be added, removed, or renamed. You can also change the style and numeric formats.

When you hover over a visualization in a graph, you see content in the hover label for that marker. A grid entry is text that appears on the hover label by default, either from the label columns or from the columns associated with the visualization. By default, the column names that are assigned to roles (and their corresponding values) are shown, along with columns that are labeled. You can customize these column names or values or add other content. Here are a few things you can do:

- add or remove a column name and its value
- rename a grid label
- add a new grid entry with a dynamic value and an optional click action. For example, you can insert an entry named "See also" with a clickable link to a Wikipedia page as the value.
- applying a style to text (for example, formatting text in bold)
- formatting numeric values (for example, setting the number of decimal places to 2)

Figure 12.46 shows adding Genre, Name, and Length in minutes, reformatting the minutes, and formatting "Genre" in bold.

Figure 12.46 Customized Gridlet

Example of Customizing a Gridlet

The following script creates the graph and hover label shown in Figure 12.46.

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA/Movie Inventory.jmp" );
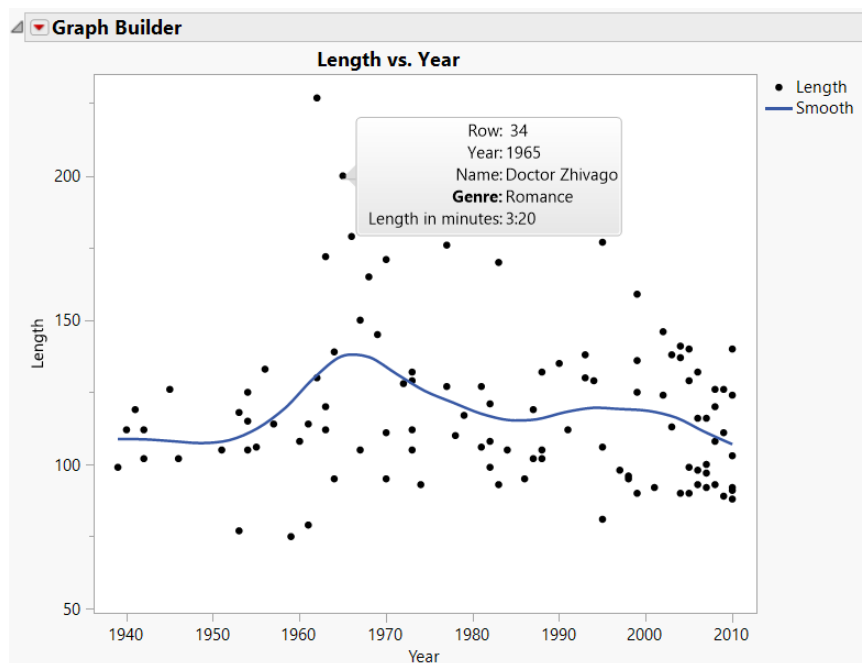
// launch the baseline visualization
dt << Graph Builder(
  Size( 534, 456 ),
  Show Control Panel( 0 ),
  Variables( X( :Year ), Y( :Length ) ),
  Elements( Points( X, Y, Legend( 3 ) ), Smoother( X, Y, Legend( 4 ) ) ),
  SendToReport(
    Dispatch(
      {},
      "Graph Builder",
      FrameBox,
      {Set Gridlet(
        // delete the Length column
        Expunge( {{Matcher( "Length" )}} ),
        // add the Name, Genre, and Length in minutes columns
        Annex(
          {{Matcher( "Name" ), value( :Name[local:_firstRow] )},
          {Matcher( "Genre" ), value( :Genre[local:_firstRow] )},
          {Matcher( "Length in minutes" ), value( local:_Length * 60 )}}
        ),
        Reformat(
        // reformat the Length in minutes format
          {{Matcher( "Length in minutes" ), Format( "hr:m", 80 ), 80}}
        ),
        Style(
        // format "Genre" in bold nine-point text
          {{Matcher( "Genre", Scope( Name ) ), Font("Segoe UI", 9,
            "Bold")
          }}
        )
      ), Add Pin Annotation(
          Seg( Marker Seg( 1 ) ),
          Index( 33 ),
          Index Row( 33 ),
          UniqueID( 610643713 ),
          FoundPt( {508, 199} ),
          Origin( {1964.8951552795, 199.00456244898} ),
          Tag Line( 1 )
      )}
    )
```

```
        )
    );
```

Gridlet Commands

The first argument in all `Set Gridlet` commands is a matcher, which identifies its scope of action (the grid entries it affects). A matcher can refer to the following:

- A single grid entry: by name, matching against a string. A grid entry is text that appears on the hover label by default, either from the label columns or from the columns associated with the visualization.
- Multiple grid entries: by column reference, regular expression or using the argument `All`.

For example, a grid entry for Mean(height) can be matched against the following:

- the name `"Mean(height)"`n
- the column reference `:height`
- the regular expression `PatReg("^.*\(height\)$")`
- the special argument `All`, which always matches all rows in the grid.

If there is a match, the command-associated action is carried against the matching entries using the arguments.

The exception is the `Annex` command, which adds a new entry to the grid if there are no matches with existing entries (to avoid unplanned naming collisions).

Example of Adding and Deleting Columns

This example describes how to use the `Expunge` command to remove the first entry added to the hover label when the underlying table has no Label columns, followed by two instances of the `Annex` command to add new rows. Note that you could have used a single `Annex` instance and passed to it a list of lists instead.

Figure 12.47 shows the original hover label and the updated hover label. Notice that Row was removed from the label. Name and Genre were added.

Figure 12.47 Example of the Expunge and Annex Commands

original hover label   updated hover label

The following script creates the updated hover label shown in Figure 12.47.

```
Names Default To Here( 1 );
```

```
dt = Open( "$SAMPLE_DATA/Movie Inventory.jmp" );
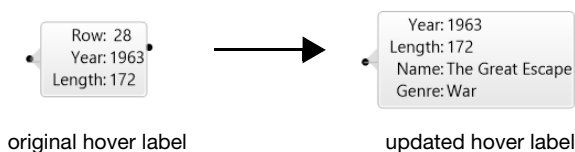
// create the baseline visualization
gb = dt << Graph Builder(
        Size( 526, 454 ),
        Show Control Panel( 0 ),
        Variables( X( :Year ), Y( :Length ) ),
        Elements( Points( X, Y, Legend( 3 ) ), Smoother( X, Y, Legend( 4 ) ) )
    );

rpt = gb << Report();
box = rpt[Framebox( 1 )];
box << Set Gridlet(
    // delete Row
     Expunge( {{Matcher( 1 )}} ),
     Annex(
        // add the Name and Genre columns to the hover label
        {{Matcher( "Name" ), value( :Name[local:_firstRow] )},
         {Matcher( "Genre" ), value( :Genre[local:_firstRow] )}}
        )

);
```

Annex (Append)

`Annex({`*`matcher`*`, `*`value`*`(<script>), <Click(<script>)>})` adds a new row to the hover label grid. The `Annex` command corresponds to the Append tab in the Hover Label Editor Gridlet panel.

This command takes three arguments:

- `Matcher`: The new name. `Matcher` should not match existing names.

- `Value`: Required. The script is evaluated in the Hover Label Execution Context. The result is the value that appears for the new row. See "Work with the Hover Label Execution Context" on page 713 for details.

  ```
  box << Set Gridlet(
      Annex( { "LDL, HDL", value(
      Char(:LDL[local:_firstRow]) || ", " || Char(:HDL[local:_firstRow])
      ) })
      );
  ```

- `Click`: Optional. `Click` is evaluated in the Hover Label Execution Context when the user clicks on annexed entry value. This option is equal to the `Click` attribute in graphlets. If present, the value portion of the new entry is rendered using a link style (blue text color with an underline) to indicate that it can be clicked.

Expunge (Delete)

Expunge({Matcher}) deletes the matcher. The Expunge command corresponds to the Delete tab in the Hover Label Editor Gridlet panel.

All matching entries are removed from the grid.

```
box << Set Gridlet( Expunge( {"age"} ) );
```

Rename

Rename({*Matcher*, Value(<string>)}) replaces the name portion of the matched entry or entries with the given value. The Rename command corresponds to the Rename tab in the Hover Label Editor Gridlet panel.

```
box << Set Gridlet( Rename( {Matcher( "name" ), Value( "students" )} ), );
```

Reformat

Reformat({*Matcher*, <*format*>}) takes a matcher and an argument that specifies a format to be applied to the matched row (or rows) grid numerical value or values. Matching entries are changed to reflect the specified format.

```
box << Set Gridlet( Reformat( {:height, Format( "Scientific", 10, 4 )} ) );
```

Style

Style({*Matcher*, <*style1*>, <*style2*>, ...}) takes a matcher and one or more style arguments. Here are the supported styles:

• Background Color

• Text Color

• Font (which further supports font family, size and style)

• Justification (left, center, or right)

• Matching entries are changed to reflect the specified styles.

```
box << Set Gridlet(
  Style(
    {{Matcher( "sex" ), Text Color( "Red" ), Font("Papyrus", 14,
    "Bold Underline")}}
  )
);
```

Create Drill-Down Graphs

A drill-down graph is created when you click on the thumbnail. Clicking on the hover graph opens it in a new window, enabling you to see the graph in more detail and also make modifications such as changing the data filter. For example, click a cell on a treemap to see details about the data in a new window (Figure 12.48). Press Ctrl and click to replace the contents of the last launched window instead of opening a new window.

- The Next in Hierarchy column property enables you to define the behavior of categorical drill downs in preset graphs. This column property defines only one level of the drill down, from a category in a grouping role in the current visualization to the category in the grouping role in the next one, launched by the graphlet.

- If the Next in Hierarchy column property is not defined, the list of categorical or nominal columns from the data table that have not been added to a filter is used.

Notes:

- The Next in Hierarchy column property is not supported in graphlets defined using Paste Graphlet. See *Using JMP* for details about the Paste Graphlet command.

- If there is no measurement, JMP tries to get one from the data table and uses Count (or no measurement) if there are no numeric columns in the data table.

Figure 12.48 shows drilling-down on a cell in a treemap.

Figure 12.48 Example of a Drill-Down Graph on a Hover Label

The following script creates the simplified versions of the treemap presets shown in Figure 12.48.

```
Names Default To Here( 1 );
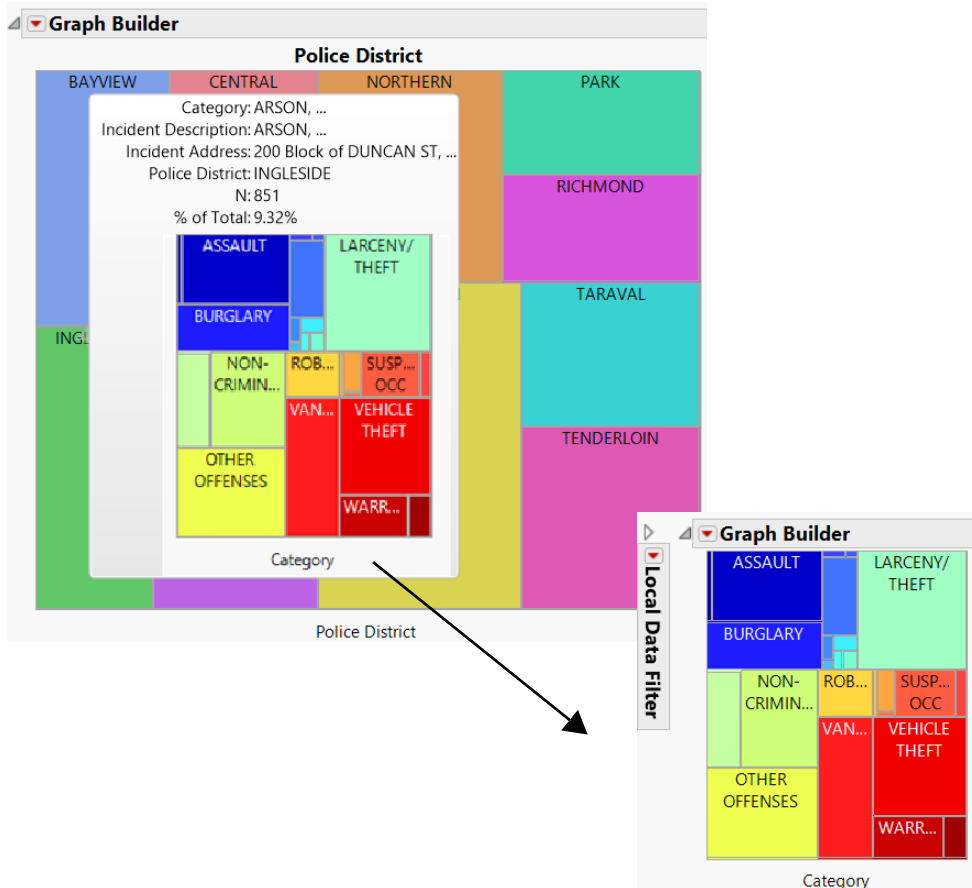
/* traverse the given list of column names, defining the Next in Hierarchy
   column property so that it creates a chain to enable the drill down */
setHierarchy = Function( {dt, cols},
    {Default Local},
    nCols = N Items( cols );
    If( nCols < 2,
        Write( "\!NsetHierarchy: Insufficient number of columns!" ),
        col = Column( dt, cols[1] );
        For( i = 2, i <= nCols, i++,
```

```
        nextCol = cols[i];
        Try(
           Column( dt, nextCol ),
           Write( "Invalid column name: " || nextCol );
           Throw();
        );
        // Eval() is needed for now
        col << Set Property( "Next in Hierarchy", Eval( nextCol ) );
        nih_name = col << Get Property( "Next in Hierarchy" );
        col = Column( dt, nextCol );
      );
   );
);

// open San Francisco Data.jmp
dt = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );

// add a Next in Hierarchy chain to the data table
setHierarchy( dt, {"Police District", "Category", "Incident Description"} );

/* Graph Builder treemap based on the first level of our hierarchy, "Police
   District" */
gb = dt << Graph Builder(
   Size( 495, 440 ),
   Show Control Panel( 0 ),
   Variables( X( :Police District ) ),
   Elements( Treemap( X ) )
);

// get a handle to the graph framebox
rpt = gb << Report;
frame = rpt[Framebox( 1 )];

/* configure the graphlet -- a hover label extension that adds a visualization
   thumbnail to the hover label. Clicking on the thumbnail launches the
   associated visualization in its own window (or replaces the graph in the
   current window on a Ctrl-click). */
frame << Set Graphlet(
   // define the thumbnail and launch treemap
   Picture(
      // next-level grouping is retrieved from the Next in Hierarchy property
      local:propColName = local:_groupings[1] << GetProperty( "Next In
   Hierarchy" );
      local:nextGrouping = Column( local:_dataTable, local:propColName );

      // define the Graph Builder graph for the thumbnail
```

```
        Graph Builder(
           Size( 200, 250 ),
           Show Legend( 0 ),
           Show Title( 0 ),
           Show Footer( 0 ),
           Show Control Panel( 0 ),
           Variables( X( local:nextGrouping ) ),
           Elements( Treemap( X ) )
        );
   )
);

// add a pinned annotation to display the graphlet
frame << Add Pin Annotation(
   Seg( TreeMapSeg( 1 ) ),
   Index( 2 ),
   Index Row( 7033 ),
   UniqueID( 581248018 ),
   FoundPt( {52, 248} ),
   Origin( {27.6544421487603, 200.14987244898} ),
   Tag Line( 1 )
);
```

Work with the Hover Label Execution Context

Hover Label Execution Context variables define the environment in which your graphlet and textlet scripts are executed.

Local Data Filter

When defining a graphlet, you specify how to visualize the data that appears as a thumbnail. But how do you specify which rows should be displayed?

This information is specified indirectly you hover over different visual elements of the baseline graph (the graph to which a graphlet definition was added). Suppose that the baseline graph is a bar chart. Hovering over one of the bars indicates that all the rows behind that bar (and only those rows) should be included in the generation of the visualization. This visualization shows up as a thumbnail on the hover label. In other words, the hovered visual element provides the data context in which the visualization is generated.

This data context is expressed in a where clause that is computed by JMP based on:

- the type of graph used in the baseline visualization
- the columns assigned to its different roles
- the current values for the particular visual element being inspected

The resulting where clause is then used to create a local data filter that is assigned to the platform specified in the graphlet definition.

The following graph roles are supported as sources for the creation of local data filters:

- X, including nested columns
- Y, if not representing a summary statistic or other calculation
- overlays

Numeric columns are used in comparison to a single value or checked for inclusion in a range, depending on the underlying visualization type. Visualizations that group numeric data in bins such as histograms and heat maps (and the box in box plots) result in the use of corresponding ranges in the where clause.

If present, the where clause of a Local Data Filter associated with the baseline visualization is captured as well.

Local Scope Context

The JSL expressions that configure different aspects of a graphlet and other hover label extensions (such as the textlet `Setup` and the gridlet `Append` and `Click` scripts) are executed inside a local namespace. This namespace is initialized with local variables that reveal information from the execution context: the hover label grid entries and references to the underlying visual element and associated data. These variables are used extensively by the graphlet presets.

Figure 12.4 describes variables that are provided in a graphlet local namespace. See "Complete Example of the Hover Label Execution Context Variables" on page 719.

Note: To see the variables at work, right-click on a graph, select **Hover Label Editor**, select the Graphlet panel, and then select one of the presets. The code that appears in the Picture field illustrates how many of the Hover Label Excecution Context variables are used to control the graphlet's behavior.

Table 12.4 Hover Label Execution Context Variables

Variable	Description

_mode	A string indicating the current evaluation context. Values include Picture, Click, Textlet, Graphlet, and Other. You can use _mode to customize a Graph Builder expression to show more or less information.
	
	From the JMP log:
	
	    {"_mode", "Textlet"},
	
	Example from the Bar preset:
	
	<pre>If(local:_mode == "Picture",
	    local:sizeDef = "Size(200, 250),";
	    local:l = 0;
	,
	    local:sizeDef = "";
	    local:l = 1;
	);</pre>
	Note: Other means the other attributes that take expressions (for example, SkipFiltersFlag).

_groupings	A list of the current grouping (X or overlay roles) data column references. See "Example of Defining the Context of a Scatterplot Matrix and Ellipse" on page 718. {"_groupings", {:sex}}
_measurements	A list of the current measurements (Y role) data column references. {"_measurements", {:height}}
_filters	A list of the current local data filter data column references. {"_filters", {:sex}}

Table 12.4 Hover Label Execution Context Variables *(Continued)*

Variable	Description
_where	A textual representation of the where clause associated with the hovered visual element. The expression is parsed to be used with JSL statements that take where clauses such as `Get Rows Where`. `_where` is useful to cache complex calculations that are based on context.

```
expr = Parse( local:_where );
r = local:_dataTable << Get Rows Where( expr
  );
ss = dt << Subset( Private, Invisible, Rows(
  r ) );
Summarize( ss, exg = By( :sex ), exm = Mean(
  :height ) );
```

Another example:

```
{"_where", ":sex == \!"F\!""}
```

Variable	Description
_whereExpr	The where clause that is associated with the hovered visual element as a JSL expression.

```
local:_h = Round(Mean(
  :height[local:_dataTable << Get Rows
  Where( local:_whereExpr )] ), 2);
```

Another example:

```
{"_whereExpr", :sex == "F"}
```

Variable	Description
_displaySegName	The class name of the underlying seg. Useful when creating descriptive text or in the definition of graph-dependent drill-down logic.

```
Title("Graphlet for " ||
  local:_displaySegName ||
      ", depth is " || Char(
  local:_drillDepth ))
```

Another example:

```
{"_displaySegName", "BarSeg"}
```

Variable	Description
_dataTable	A reference to the underlying data table. Useful for creating subsets. See "Add Rich Text to Hover Labels Using Textlets" on page 692 for an example.

```
{"_dataTable", Data Table( "Big Class" )}
```

Table 12.4 Hover Label Execution Context Variables *(Continued)*

Variable	Description
_summaryStatistic	A string with the current summary statistic. Useful for passing to new graphs in a drill down. {"_summaryStatistic", "Mean"}
_drillDepth	An integer that indicates how many drill-down levels have been opened before. Useful to limit drill-down chains or to control SkipFilters logic. See "Example of Defining the Context of a Scatterplot Matrix and Ellipse" on page 718 {"_drillDepth", 1}
_firstRow	The 1-based index of the first row associated with the hovered visual element. See "Example of Adding and Deleting Columns" on page 707. {"_firstRow", 1}
_underlyingRows	The count of rows associated with the visual element that appears in the hover label. You can use this variable to decide the type of the visualization based on the number of rows to display, going from the low granularity of a heat map to a dense marker plot if the number of rows is small enough (and vice versa). See "Reapply" on page 704 for an example. {"_underlyingRows", 18}

In addition to these variables, a variable is defined for each of the hover label grid entries. The name of this variable is taken for the grid entry name, with a '_' prefix.

Note: See "Example of Defining the Context of a Scatterplot Matrix and Ellipse" on page 718 for examples of these variables.

As an example, using Hover Label Execution Context variables, you can create a scatterplot matrix, hover over a point to show a close-up ellipse, and then click the ellipse to modify it in Graph Builder. The graphlet shows the same visualization as the one in the underlying scatterplot matrix cell (a scatterplot of the two columns associated with that cell).

Clicking on the thumbnail then has the effect of "extracting" the cell from the matrix and opening it up as an independent visualization.

The script shown in "Example of Defining the Context of a Scatterplot Matrix and Ellipse" on page 718 enables the user to extract the matrix cells in the main scatterplot matrix.

Figure 12.49 Viewing an Ellipse in a Scatterplot Matrix

Example of Defining the Context of a Scatterplot Matrix and Ellipse

The following script shows an example of using the context in which the script runs. This script creates the scatterplot matrix shown in Figure 12.49.

```
Names Default To Here( 1 );
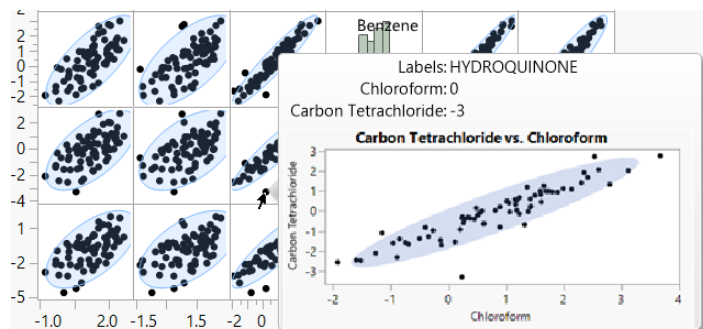
dt = Open( "$SAMPLE_DATA/Solubility.jmp" );
win = New Window( "Scatterplot Matrix Graphlet Example",
    plt = dt << Multivariate(
        Y(
            "1-Octanol"n,
            :Ether,
            :Chloroform,
            :Benzene,
            :Carbon Tetrachloride,
            :Hexane
        ),
        Estimation Method( "Row-wise" ),
        Scatterplot Matrix(
            Density Ellipses( 1 ),
            Shaded Ellipses( 1 ),
            Ellipse Color( 69 ),
            Horizontal( 1 )
        )
    )
);
frame = win << xpath( "//FrameBox" );
frame << Set Graphlet(
    Picture(
        Show( local:_displaySegName );
```

```
              If( local:_displaySegName == "MarkerSeg",
                // scatterplot cells
                Graph Builder(
                   Size( 400, 200 ),
                   Show Control Panel( 0 ),
                   Show Legend( 0 ),
                   Variables(
                      X( local:_groupings[1] ),
                      Y( local:_groupings[2] )
                   ),
                   Elements( Points( X, Y ), Ellipse( X, Y ) )
                ),
                // else histograms in matrix diagonal
                If( local:_N > 5, /* Note: only present if baseline graph
                   is a histogram.*/
                   Graph Builder(
                      Size( 400, 400 ),
                      Show Control Panel( 0 ),
                      Show Legend( 0 ),
                      Variables( X( local:_groupings[1] ) ),
                      Elements( Histogram( X ) )
                   ),
                   Graph Builder(
                      Size( 400, 400 ),
                      Show Control Panel( 0 ),
                      Show Legend( 0 ),
                      Variables( X( local:_groupings[1] ) ),
                      Elements( Points( X ) )
                   )
                )
             );
          ),
          // skip filters if you are at the first drill level -- the original graph
          SkipFilters( local:_drillDepth == 1 ), // no list -- all filters
          Title( "Graphlet" ),
          Reapply( local:_displaySegName == "HistSeg" & local:_N > 5 )
       );
```

Complete Example of the Hover Label Execution Context Variables

The following script produces the graph shown in Figure 12.50. When you hover over a bar,
the hover label appears. Details about the data and the Hover Label Context Execution
variables appear in the log.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Graph Builder(
   Size( 528, 450 ),
```

```
Show Control Panel( 0 ),
Variables( X( :sex ), Y( :height ) ),
Elements( Bar( X, Y, Legend( 3 ) ) ),
SendToReport(
   Dispatch(
      {},
      "Graph Builder",
      FrameBox,
      {Set Textlet(
         Setup(
            local:encodedWhere = XML Encode( local:_where );
            /* Show() provides additional information about the
            HLEC variables in the log */
            Show( Namespace( "local" ) << Get Contents() );
         ),
         Markup(
            "
<b>Groupings</b>: {local:_groupings}
<b>Measurements</b>: {local:_measurements}
<b>Summary statistic</b>: {local:_summaryStatistic}
<b>Filter columns</b>: {local:_filters}
<b>Where clause</b>: {local:encodedWhere}
<b>Graph type</b>: {local:_displaySegName}
<b>Data table</b> {local:_dataTable}
<b>Drill depth</b>: {local:_drillDepth}
<b>First Row</b>: {local:_firstRow}
<b>Underlying Rows</b>: {local:_underlyingRows}
"
         )
      )}
   )
)
);
```

Figure 12.50 Bar Chart with Textlet

Three-Dimensional Scenes

Scripting in Three Dimensions

JSL includes commands for scripting three-dimensional scenes derived from OpenGL. Although not a complete OpenGL$^{®}$ implementation, JSL's 3-D scene commands enable complex, interactive plots to be constructed and viewed. The Surface Plot platform in JMP is built using JSL scene commands.

Figure 13.1 Sample Three-Dimensional Shapes

Contents

About JSL 3-D Scenes

JMP's 3-D scene language is built on top of the OpenGL API, extending, replacing, and leaving out various parts of the OpenGL API. As such, the 3-D scene language is not certified or licensed by Silicon Graphics, Inc. under the OpenGL API.

This chapter documents JMP's JSL commands for creating 3-D scenes but is not a tutorial on OpenGL programming. If you are not familiar with OpenGL programming, you might want supplemental material. If you are familiar with OpenGL programming, you still need to read this chapter because some items are nonstandard.

JMP is shipped with sample files in the Scene3D subfolder of Sample Scripts to get you started and give you some ideas. Some of the example scripts are similar to some of the examples in this chapter; some are almost complete applications.

The website https://opengl.org is a good source for information, as is your favorite search engine.

JMP's Scene 3-D language does some work for you that the OpenGL API requires you to do for yourself. JMP makes text easy, gives you a built-in arcball controller, and makes sure the matrix operations that belong on the model view stack and projection stack go on their respective stacks. JMP uses its own display list manager so that your scenes can be journaled and played back later, and provides a select mechanism that calls back to your JSL code to tell you what object in your scene is under the mouse, with almost no extra effort on your part. At this time, JMP does not provide access to some features, like texturing.

JSL 3-D Scene Boxes

These commands are necessary to set up and configure a 3-D scene.

Like all displays in JMP (detailed in the "Display Trees" chapter on page 517), 3-D scenes must be placed in a display box (in this case, a Scene Box). This box is then placed in a window. Therefore, a simple 3-D scene script has the following form.

```
myScene = Scene Box(300, 300); // create a 300 by 300 pixel scene box
...(commands to set up the scene)...
New Window("3-D Scene", myScene); // draw the scene in a window
...(commands that manipulate the scene)
```

The scene can be sent messages that construct elements in the scene. Typical messages alter the viewer's vantage point, construct physical elements in the scene itself, or manipulate lights and textures. These messages are maintained in a display list and are manipulated in one of two ways:
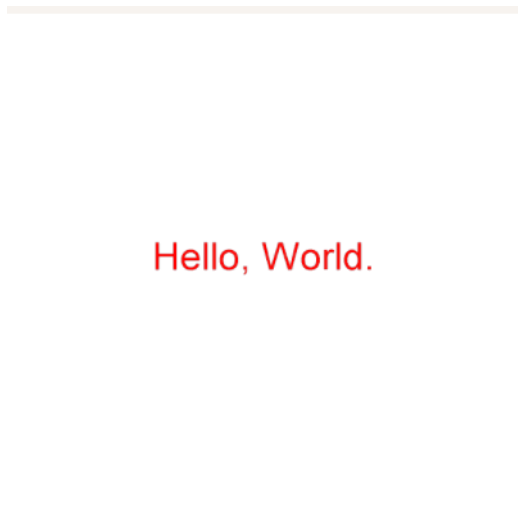
- They are sent as messages to the scene, which immediately adds them to the scene's internal display list.

- They are sent as messages to a display list stored in a global variable, which is called by the scene's display list later.

Figure 13.2 Constructing Scenes

```
myscene=Scene Box(300,300)
dl=Scene Display List
```


For example, of commands sent to the scene's display list directly, consider the following small script. Each of the commands is explained in detail later in the chapter.

```
scene = Scene Box( 400, 400 ); // make a scene box
New Window( "Example 1", scene ); // put the scene in a window
scene << Perspective( 45, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -4.5 ); // move to (0,0,-4.5) to draw
scene << Color( 1, 0, 0 ); // set the RGB color of the text
scene << Text( center, baseline, 0.2, "Hello, World." ); // add text
scene << Update; // update the scene
```

The first two lines create a scene and place it in a window. The Perspective command defines the viewing angle and field depth. By sending it as a message to the scene, it is immediately added to the scene's display list. Because the "Hello World" text is drawn at the origin (0, 0, 0), the Translate command is added to the display list. This command moves the camera back a bit so that the origin is in the field of vision. The color is set to red with the Color command, the text is drawn, and the Update command causes the scene to be rendered (in other words, causes the display list that contains the commands to be drawn.)

Figure 13.3 Hello World

Equivalently, the commands to construct the display can be accumulated in a display list
stored in a global variable, which is then sent to the scene all at once. To define a global
variable as a display list, assign it using the Scene Display List function. For example, to
use the global greeting as a display list, issue the command

```
greeting=Scene Display List();
```

Display commands can then be sent as messages to greeting. An equivalent "Hello World"
example using a display list follows.

```
greeting = Scene Display List();
greeting << Color( 1, 0, 0 ); // set the RGB color of the text
greeting << Text( center, baseline, 0.2, "Hello, World." ); // add text

// draw the window and send it the stored display list
scene = Scene Box( 400, 400 ); // make a scene box
New Window( "Example 1", scene ); // put the scene in a window
scene << Perspective( 45, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -4.5 ); // move to (0,0,-4.5) to draw
scene << Call List( greeting ); // send the display list to the scene
scene << Update; // update the scene
```

Note which commands were separated into the display list, and which were applied to the
scene directly. Those that manipulate the camera (Translate and Rotate) are applied to the
scene. Those that define the object (Color and Text) were relegated to the display list. This is
done so that the display list can be called many times to replicate the object at different
positions.

Setting the Viewing Space

3-D scenes can be rendered in two ways. *Orthographic* projections place the elements in a box, where coordinates are not changed to accommodate the perspective of the viewer. *Perspective* projections modify the display to simulate the position of the elements in relation to the position of the viewer. For example, two parallel lines (like railroad tracks) stay parallel in orthographic projections, but seem to connect at a distance in perspective projections.

Figure 13.4 Parallel Lines in an Orthographic Projection (left) and a Perspective Projection (right)

As another example, imagine looking at a tube edge-on (like a telescope). In an orthographic projection, the tube would appear as a thin circle. In a perspective projection, the circle would have a thickness; the hole at the far end of the tube would appear smaller than the close hole, and the interior of the tube is visible.

Therefore, the viewable space of an orthographic projection is a rectangular shape. That of a perspective projection is the frustum of a pyramid (that is, a pyramid whose top has been sliced off).

Figure 13.5 Comparing Projections

Orthographic Scene

Perspective Scene

In general, perspective projections give a more realistic view of the world, since it mimics the way an eye or a camera sees. Orthographic projections are important when it is essential to preserve dimensions, such as an architectural CAD program.

Setting Up a Perspective Scene

To set up a perspective scene in JSL, send the `Perspective` command to a display list.

```
Perspective (angle, near, far)
```

where *angle* is the viewing angle, *near* is the distance to the near plane, and *far* is the distance to the far plane, as illustrated in the drawing above. A couple of things need to be remembered when defining the viewing space.

• Items outside the viewing space (for example, closer than the *near* plane or farther than the *far* plane) are not drawn. They are clipped off.

• The ratio of *far* to *near* needs to be small. This enables the rendering engine to effectively determine which items should be drawn "on top of" other items, simulating closeness of items. The *near* argument must be greater than zero.

The "Hello World" example contains the line

```
scene << Perspective( 45, 3, 7 ); // define the camera
```

This defines a 45-degree viewing angle, with a near plane 3 units from the viewer and a far plane 7 units from the viewer.

The viewing angle functions in the same way as a wide angle or telephoto lens on a camera. Small viewing angles zoom into a drawing; wide angles zoom out. In other words, a small viewing angle maps the screen space onto a small portion of the scene, resulting in apparently larger scene elements. A large viewing angle maps the screen space onto a large portion of the scene, resulting in apparently small screen elements. The size of scene elements can therefore be manipulated using the *angle* argument of the Perspective function. The picture here shows the hello world script with perspective angles of 45 and 90 degrees.

Figure 13.6 Changing the Perspective

Hello, World.
```
scene << Perspective( 45, 3, 7 );
```

Hello, World.
```
scene << Perspective( 90, 3, 7 );
```

As an alternative to the Perspective command, you can define the actual viewing frustum with the Frustum command.

```
Frustum(left, right, bottom, top, near, far);
```

The frustum's viewing volume is defined by (*left, bottom, near*) and (*right, top, near*), which specify the (*x, y, z*) coordinates of the lower left and upper right corners of the near clipping plane; *near* and *far* give the distances from the viewpoint to the near and far clipping planes.

Setting up an Orthographic Scene

Orthographic scenes are specified in ways similar to perspective scenes. Issue the command

```
Ortho(left, right, bottom, top, near, far)
```

which specifies the four corners of the near plane, the distance to the near plane, and the distance to the far plane.

If you are dealing with a simple 2-D environment, you can set up a two-dimensional orthographic scene with the command

```
Ortho2D (left, right, bottom, top)
```

which specifies the corners of the two-dimensional view.

Changing the View

One of the advantages of creating a 3-D scene is the ease that they can be viewed from different angles and positions. The `Translate` and `Rotate` commands let you set the position from which you view the scene.

In addition, you can use the `ArcBall` command to enable the user to change the viewing angle interactively.

The Translate Command

You have actually seen the `Translate` command in earlier sample scripts. It sets the position from which the scene is viewed. The arguments give the amount to move from the current position in the x, y, and z direction.

```
Translate (x, y, z)
```

For example,

```
Translate( 0.0, 0.0, -2 );
```

moves the origin two units in the negative z direction.

Initially, the origin and camera were at the same place. Now, the camera can see the origin because the camera faces down the negative z axis.

The Rotate Command

The `Rotate` command is used to modify the viewing angle of a scene. It has the following format.

```
Rotate (degrees, xAxis, yAxis, zAxis)
```

This rotates by degrees around the axis described by the vector (*xAxis*, *yAxis*, *zAxis*). For example, to rotate a model 90 degrees about the x-axis, use `Rotate( 90, 1, 0, 0 )`.

You can also specify the three axis values in a matrix. For example, `Rotate( 90, [1, 0, 0]`).

Note: The `Rotate` command uses degrees, in contrast to JMP's trigonometric functions, which use radians.

`Translate` and `Rotate` are also used to position objects with respect to each other. The first `Translate` or `Rotate` can be thought of as positioning everything that follows with respect to the camera. Subsequent `Translate` and `Rotate` commands are used to position objects, such as spheres, cylinders, disks, and display lists in `Call List` and `ArcBall` commands. For example, suppose you have a display list named `table` and another named `chair`. Your scene might look like this:

Figure 13.7 Using Translate and Rotate

```
scene << Perspective(...);    ⎫
scene << Look At (...);       ⎬————————— Set up scene

scene << Call List (table);   ⎬————————— Draw table

scene << Translate(...);      ⎫
scene << Rotate(...)          ⎬————————— Position and draw first chair
scene << Call List (chair);   ⎭

scene << Translate(...);      ⎫
scene << Rotate(...);         ⎬————————— Position and draw second chair
scene << Call List (chair);   ⎭
```

The following example uses the `Rotate` command inside a `For` loop to continuously change the viewing angle of a scene. It draws a cylinder that swings around a central point. This central point is shown by a small sphere.

```
scene = Scene Box( 600, 600 ); // make a scene box...holds an OpenGL scene.

New Window( "Example 1", scene ); // put the scene in a window

For( i = 1, i < 360, i++,
    scene << Clear;

    // the lens is 50 degrees, near is 1 units from the camera, far is 10.
    scene << Perspective( 50, 1, 10 );
    scene << Translate( 0.0, 0.0, -2 );

    scene << Rotate( i, 1, 0, 0 );
    scene << Rotate( i * 3, 0, 1, 0 );
    scene << Rotate( i * 3 / 2, 0, 0, 1 );
```

```
        scene << Color( 0, 1, 0 ); // green for cylinder
        scene << Cylinder( 0.5, 0.5, 0.5, 40, 10 );
        scene << Color( 0, 0, 0 ); // black for sphere
        scene << Sphere( 0.01, 10, 5 );
        scene << Update;
        Wait( 0.01 ); );
```

Figure 13.8 Rotating a Cylinder

Note the use of the Update command at the end of the scene messages. This command tells JMP to make the displayed screen agree with the current state of the display list. It is important to clear the list at the beginning (so that the list does not contain the old angles as well as the current) and update the scene after each change.

The Look At Command

The Look At command is an alternative way to set the camera view.

```
    Look At( eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )
```

The Look At command puts the camera at the *eye* coordinates and points it toward the *center* coordinates. The *up* vector describes how the camera is rotated on its line of sight. The model is typically constructed at the origin. Therefore, a JMP scene should have either a Look At or a Translate command near its beginning to move the camera away from the origin.

First, clear the scene box of any commands from the previous frame.

```
scene << Clear;
```

Then use one of these projections:

```
scene << Perspective( 45, 2, 10 );
scene << Frustum( -.5, .5, -.5, .5, 1, 10 );
scene << Ortho( -2, 2, -2, 2, 1, 10 );
scene << Ortho2d( -2, 2, -2, 2 );
```

Note: If you use the `ortho2d` projection, you should not also set the camera position using either `Translate` or `Look At`.

Finally, use either `Translate` or `Look At` to set the camera position:

```
scene << Translate(0.0, 0.0, -4.5);
/* the camera faces down the negative Z axis.
   move it back so that 0,0,0 is in view. */
scene << Look At( /*eye*/ 3,3,3, /*center*/ 0,0,0, /*up*/ 1,0,0 );
/*this is much easier. */
```

Once the scene and camera position are set, add your model.

The ArcBall

Sometimes you want a scene to rotate based on the movements of the mouse. The Surface Plot platform in JMP is an example of a 3-D scene that rotates based on mouse movements.

An *ArcBall* creates a sphere around the 3-D scene and enables the user to click the sphere's surface and drag it around, thus causing the scene to rotate.

Use an `ArcBall` instead of a `Call List` command to place the scene in an ArcBall. Scenes that are attached to an ArcBall automatically respond to clicks and drags of the mouse. Custom programming is not needed. However, rotations made in the arcball are not saved. (Technically, the ArcBall is surrounded by an implicit `Push Matrix` and `Pop Matrix` block, so the movements are gone after it returns. See "Using the Matrix Stack" on page 748 for more information about pushing and popping.)

For example, examine the script from "Primitives Example" on page 738. Change the single line

```
scene << CallList(shape); // send the display list to the scene
```

so that it reads

```
scene << ArcBall(shape,2); // send the display list to an arcball
```

This displays the script with an associated arcball with diameter 2. When you run the script and the window appears, right-click and select **Show ArcBall > Always** from the menu that appears.

Note: ArcBall comes from an article by Shoemake (1994) found in *Graphics Gems IV*, published by Academic Press.

This sets the display so that the ArcBall is always showing. Click and drag on the ArcBall to rotate the scene. The right-click menu with Background Color, Use Hardware Acceleration, and Show ArcBall is always available, whether the scene is displayed through a platform, in a journal, or through JSL.

Note: The ArcBall does not have to be showing to react to mouse commands. It is shown here for display purposes only.

You can also set the display state of the ArcBall in JSL using the Show ArcBall command.

```
scene << Show Arcball(state)
```

where *state* is During Drag, Always, or Never.

Figure 13.9 Showing the Arc Ball

Graphics Primitives

All scenes in JSL are built with a small number of graphics primitives. These fundamental elements function as the building blocks for complicated scenes.

Every graphics primitive involves specifying vertices. In some cases, the vertices are simply drawn as points. In others, the vertices are connected to form polygons. To draw a primitive, you must specify the type of primitive and the coordinates and properties of the vertices involved. In JSL, this specification is accomplished through the Begin and End statements.

```
scene<<Begin(primitive type);
...(commands specifying vertices and their properties)...
scene<<End();
```

To specify the coordinates of the vertices, use the `vertex` command.

```
scene<<Begin(primitive type);
scene<<Vertex(x, y, z);
...
scene<<End();
```

The options for *primitive type* are the following. In these examples, assume that v0, v1, and so on, have been specified between a `Begin` and `End` pair, similar to the following.

```
scene<<Begin(primitive type);
scene<<Vertex(x0,y0,z0) // specify vertex v0
scene<<Vertex(x1,y1,z1) // specify vertex v1
...
scene<<Vertex(xn,yn,zn) // specify vertex vn
scene<<End();
```

Table 13.1 Primitive Types

primitive type=POINTS	Draws a point at each of the vertices.	
primitive type=LINES	Draws a series of (unconnected) line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If n is odd, the last vertex is ignored.	
primitive type=POLYGON	Draws a polygon using the points v0,..., vn as vertices. Three vertices must exist, or nothing is drawn. In addition, the polygon specified must not intersect itself and must be convex. If the vertices do not satisfy these conditions, the results are unpredictable.	

Table 13.1 Primitive Types *(Continued)*

primitive type=TRIANGLES	Draws a series of (disconnected) triangles using vertices v0, v1, v2, then v3, v4, v5, and so on. If the number of vertices is not an exact multiple of 3, the final one or two vertices are ignored.	
primitive type=LINE_STRIP	Draws a line segment from v0 to v1, then from v1 to v2, and so on. Therefore, n vertices specify n–1 line segments. Nothing is drawn unless there is more than one vertex. There are no restrictions on the vertices describing a line strip; the lines can intersect arbitrarily.	
primitive type=LINE_LOOP	Same as LINE_STRIP, except that a final line segment is drawn from the last vertex to the first, completing a loop.	
primitive type=QUADS	Draws a series of quadrilaterals (four-sided polygons) using vertices v0, v1, v2, v3, then v4, v5, v6, v7, and so on. If the number of vertices is not a multiple of 4, the final one, two, or three vertices are ignored.	
primitive type=QUAD_STRIP	Draws a series of quadrilaterals (four-sided polygons) beginning with v0, v1, v3, v2, then v2, v3, v5, v4, then v4, v5, v7, v6, and so on. The number of vertices must be at least 4 before anything is drawn, and if odd, the final vertex is ignored.	

Table 13.1 Primitive Types *(Continued)*

primitive type=TRIANGLE_STRIP	Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. There must be at least three vertices for anything to be drawn.	
primitive type=TRIANGLE_FAN	Same as TRIANGLE_STRIP, except that the vertices are v0, v1, v2, then v0, v2, v3, then v0, v3, v4, and so on.	

Primitives Example

The following short example illustrates the use of a graphics primitive.

```
shape = Scene Display List(); // create a display list and send it commands
shape << Color( 1, 0, 0 ); // set the RGB color of the text
shape << Begin( POLYGON );
shape << Vertex( 0, 0, 0 );
shape << Vertex( 0, 3, 0 );
shape << Vertex( 3, 3, 0 );
shape << Vertex( 5, 2, 0 );
shape << Vertex( 4, 0, 0 );
shape << Vertex( 2, -1, 0 );
shape << End();

// draw the window and send it the stored display list
scene = Scene Box( 400, 400 ); // make a scene box
New Window( "Primitive", scene ); // put the scene in a window
scene << Perspective( 90, 3, 7 ); // define the camera
scene << Translate( 0.0, 0.0, -5 ); // move to (0,0,-5) to draw
scene << Call List( shape ); // send the display list to the scene
scene << Update; // update the scene
```

The first section of the script creates a display list named shape. Inside this display list, a polygon is defined using six vertices.

The second section of the script creates a scene box and a new window. It then uses the Call List message to put the list in the display.

Note that all the *z*-coordinates are zero, which makes sure the polygon lies in a plane. Polygons that do not lie in a plane can cause unpredictable results.
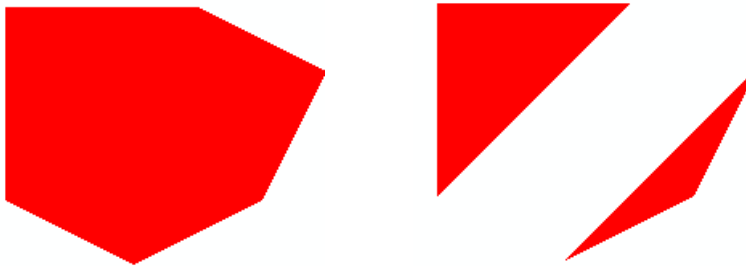
Experiment with the line

```
shape <<Begin(POLYGON);
```

by changing it to some of the other primitive types. For example, changing it to

```
shape <<Begin(TRIANGLES);
```

results in a different picture.

Figure 13.10 Polygon (left) and Triangles (right)

Controlling the Appearance of Primitives

JSL has several commands that let you tailor-make the appearance of primitive drawing objects. You can also specify the widths of lines and their stippling pattern (that is, whether they are dashed, dotted, and so on.)

Size and Width

To set the point size of rendered objects, use the Point Size command.

```
Point Size( n )
```

where *n* is the number of pixels. Note that this might not be the actual number of pixels rendered, depending on other settings such as anti-aliasing and your hardware configuration.

Set the line width using the Line Width command

```
Line Width( n)
```

where *n* is the number of pixels. The argument *n* must be larger than zero and is, by default, one.

Stippling Pattern

To make stippled lines, use the Line Stipple command.

```
Line Stipple( factor, pattern )
```

Factor is a stretching factor. *Pattern* is a 16-bit integer that turns pixels on or off. Use Enable(LINE_STIPPLE) to turn the effect on.

To construct a line stippling pattern, write a 16-bit binary number that represents the stippling pattern that you desire. Note that the pattern should read from right to left, so your representation might seem backward to the way it is rendered. Convert the binary number to an integer and use this as the *pattern* argument.

For example, imagine you want the dotted line pattern 0000000011111111. This is equal to 255 in decimal notation, so use the command Line Stipple(1, 255).

The *factor* argument expands each binary digit to two digits. In the example above, Line Stipple(2, 255) would result in 00000000000000001111111111111111.

For example, the following script draws three lines, each of different widths (the Line Width commands) and stippling patterns.

```
scene = Scene Box( 200, 200 ); // make a scene box...holds an OpenGL scene

New Window( "Stipples", scene ); // put the scene in a window
scene << Ortho( -2, 2, -2, 2, -1, 1 );
scene << Color( 0, 0, 0 ); // set the RGB color of the text

scene << Enable( LINE_STIPPLE );
scene << Line Width( 2 );
scene << Line Stipple( 1, 255 );

scene << Begin( LINES );
scene << Vertex( -2, -1, 0 );
scene << Vertex( 2, -1, 0 );
scene << End();

scene << Line Width( 4 );
scene << Line Stipple( 1, 32767 );

scene << Begin( LINES );
scene << Vertex( -2, 0, 0 );
scene << Vertex( 2, 0, 0 );
scene << End();
```

```
scene << Line Width( 6 );
scene << Line Stipple( 3, 51 );

scene << Begin( LINES );
scene << Vertex( -2, 1, 0 );
scene << Vertex( 2, 1, 0 );
scene << End();
scene << Update;
```

Figure 13.11 Stipples

Note: Stipple patterns "crawl" on rotating models because they are in screen pixels, not model units. Lines in the model change length on the screen even though nothing changes in model units.
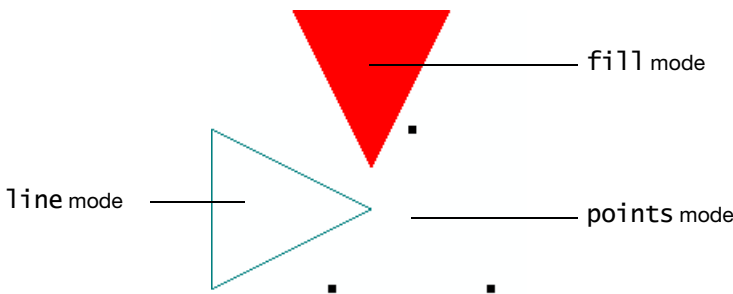
Fill Pattern

Polygons are rendered with both a front and a back, and the drawing mode of each side is customizable. This enables the user to see the difference between the back and front of the polygon.

To set the drawing mode of a polygon, use the Polygon Mode command.

```
Polygon Mode( face, mode )
```

where *face* can be FRONT, BACK, or FRONT_AND_BACK, and *mode* can be POINT, LINE, or FILL.

Figure 13.12 Points, Line, and Fill Modes

For example, the following script creates a display list that defines a triangle. This display list is used three times in conjunction with `Translate`, `Rotate`, and `Color` commands to draw triangles in three positions. In addition, the `Polygon Mode` command changes the drawing mode of each triangle. Note there is no explicit call to the `FILL` mode, since it is the default.

The following table dissects the script, showing how the `Translate` and `Rotate` commands accumulate to manipulate a single display list.

Table 13.2 Translate and Rotate Commands

Code from above script	comments
`shape = Scene Display List();`	Creates a display list

`shape << Begin( TRIANGLES );`
`shape << Vertex( 0, 0, 0 );`
`shape << Vertex( -1, 2, 0 );`
`shape << Vertex( 1, 2, 0 );`
`shape << End();`
	Creates a display list named shape that holds vertices for the triangles. All the z vertices are zero since this is a two dimensional scene

`scene = Scene Box( 200, 200 );`
`New Window( "Fill Modes", scene );`
`scene << Ortho2d( -2, 2, -2, 2 );`
	Put the scene in a display box, and create a new window.

`scene << Color(1,0,0);`
`scene << Call List( shape );`
 	
`// update the scene to see the triangle`
`scene << Update;`
	Draw the first triangle in red.

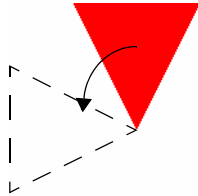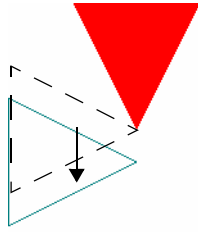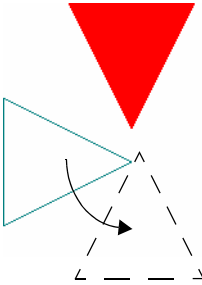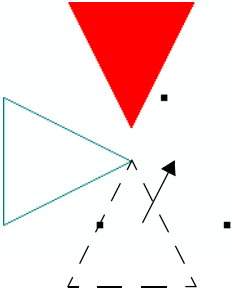**Table 13.2** Translate and Rotate Commands *(Continued)*

Code from above script	comments
| ```
scene << Rotate ( 90, 0, 0, 1 );
scene << Translate ( -0.5, 0, 0 );
scene << Color( 0, 0.5, 0.5 );
scene << Polygon Mode( FRONT_AND_BACK,
LINE );
scene << Call List( shape );

// update the scene to see the triangle
scene << Update;
``` | Draw the second triangle in teal. Note that we first rotate the triangle.<br><br><br><br>And then translate it.<br><br> |

Table 13.2 Translate and Rotate Commands *(Continued)*

Code from above script	comments
| ```
scene << Rotate( 90, 0, 0, 1 );
scene << Translate( -0.5, -1, 0 );
scene << Color( 0, 0, 0 );

// large points so that they are visible
scene << Point Size( 5 );
scene << Polygon Mode( FRONT_AND_BACK,
POINT );
scene << Call List( shape );

// update the scene to see the triangle
scene << Update;
``` | Draw the third triangle as black points. First rotate.<br><br><br><br>And then translate to get the final picture.<br><br> |

Some developers use the fill mode in concert with the line mode to draw a filled polygon with a differently colored border. However, due to the way the figures are rendered, they sometimes do not line up correctly. The Polygon Offset command is used to correct for this so-called "stitching" problem.

    Polygon Offset (*factor*, *units*)

To enable offsetting, use Enable(POLYGON_OFFSET_FILL), Enable(POLYGON_OFFSET_LINE), or Enable(POLYGON_OFFSET_POINT), depending on the desired mode. The actual offset values are calculated as $m*(factor)+r*(units)$. m is the maximum depth slope of the polygon and r is the smallest value guaranteed to produce a resolvable difference in window coordinate depth values. Start with Polygon Offset(1,1) if you need this.

An example of Polygon Offset is in the Surface Plot platform, when a surface and a mesh are displayed on top of each other, or a surface and contours displayed on top of each other. In either case, the surface would interfere with the lines if the lines were not moved closer or the surface moved farther from the viewer.

Other Uses of Begin and End

Although vertices are typically specified between begin and end statements, there are other commands that are valid. These commands are discussed in other sections of this chapter.

- `Vertex` adds a vertex to the list
- `Color` changes the current color
- `Normal` sets the normal vector coordinates
- `Edge Flag` controls drawing of edges
- `Material` sets material properties
- `Eval Coord` and `Eval Point` generate coordinates
- `Call List` executes a display list.

Drawing Spheres, Cylinders, and Disks

There are several pre-defined commands that allow for quick rendering of spheres, cylinders, and disks. The advantage of these commands is not only their ease-of-use, but that they have special lighting properties (their "normals") built in.

Construction

The following commands are used to construct cylinders, disks, partial disks, and spheres.

Cylinders

```
Cylinder( baseRadius, topRadius, height, slices, stacks )
```

baseRadius is the radius of the cylinder's base. Similarly, *topRadius* is the radius of the top. *height* is the height of the cylinder.

Slices can be 10 for a reasonably accurate cylindrical shape. Using `QuadricNormals(Smooth)` helps the appearance.

Stacks sets the number of vertices available for lighting reflections. Use a larger value for *Stacks* for accurate "hot-spots".

Disks

The following command draws a paper-thin disk with an *innerRadius* hole in the middle.

```
Disk( innerRadius, outerRadius, slices, loops )
```

Like `Cylinder`, *slices* controls the accuracy of the curve and *loops* makes more vertices (for lighting accuracy).

```
Partial Disk( innerRadius, outerRadius, slices, loops, startAngle,
    sweepAngle )
```

The `Partial Disk` command works like `Disk`, but with a slice of the disk removed. Specify the part of the disk that is showing using *startAngle* and *sweepAngle*.

Spheres

The following command draws a sphere with the specified *radius*.

```
Sphere( radius, slices, stacks )
```

The *slices* can be thought of as longitudes and *stacks* as latitudes. About 10 of each make a nicely drawn sphere.

Lighting

It is not necessary to make specific calculations of normal vectors (as is the case for customized surfaces) for spheres, disks, and cylinders. However, you can use the following commands to customize the automatic lighting.

`Quadric Normals(`*mode*`)` tells what type of normal should be automatically generated. The argument *mode* can be `None`, `Flat`, or `Smooth`. `Flat` makes faceted surfaces. `Smooth` makes the normals at each vertex be the average of the adjacent polygons.

`Quadric Orientation(`*mode*`)` determines which way the normals point. The argument *mode* can be `Inside` or `Outside`.

`Quadric Draw Style(`*mode*`)` specifies the drawing mode. The argument *mode* can be `Fill`, `Line`, `Silhouette`, or `Point`.

JMP uses the values that you set for `Quadric Normals`, `Quadric Orientation`, and `Quadric Draw Style` for subsequently generated cylinders, disks, and spheres.

Figure 13.13 Draw Styles

Fill   Line   Silhouette   Point

> **Note:** Other OpenGL documentation refers to quadric objects. JMP has only one and always uses it.

Drawing Text

As shown in the "Hello World" example above, text is added to a scene using the `Text` command.

    Text(*horz*, *vert*, *size*, *string*, *<billboard>*)

- *horz* can be `Left`, `Center`, or `Right` justification.
- *vert* can be `Top`, `Middle`, `Baseline`, or `Bottom` justification.
- *size* represents the height of a capital letter M in model coordinates.
- *string* is the text to draw.
- *billboard* is an optional argument that causes the text to rotate with the model. Text with this option always faces the viewer.

The font is always the JMP Text font. You can change the text font from the preferences menu, but because of the way JMP caches fonts for scenes, changes might not take effect until JMP is restarted.

> **Note:** Text is not part of the standard OpenGL definition.

Using Text with Rotate and Translate

The following example uses the text command in conjunction with the `Translate` and `Rotate` commands.

```
scene = Scene Box( 600, 600 ); // make a scene box...holds an OpenGL scene
New Window( "Example 2", scene ); // put the scene in a window
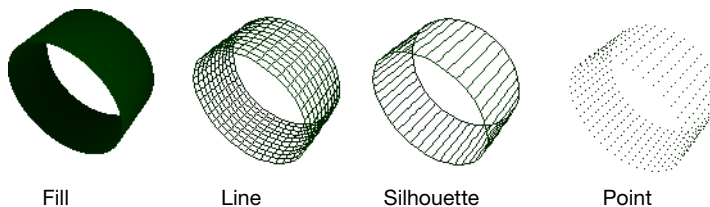
// the "lens" is 45 degrees, near is 3 units from the camera, far is 7
scene << Perspective( 45, 3, 7 );

// move the world so that 0,0,0 is visible in the camera
scene << Translate( 0.0, 0.0, -4.5 );

// rotate the first text about the Y (vertical on screen) axis
scene << Rotate( 30, 0, 1, 0 );
scene << Color( 1, 0, 2 ); // magenta
scene << Text( "center", "baseline", .2, "Top magenta string" );
```

```
// the next string is even farther away from the camera
scene << Translate( 0.0, 0.0, -2.0 );

// rotate the second text about the Y (vertical on screen) axis
scene << Rotate( 30, 0, 1, 0 );
scene << Color( 0, 1, 1); // aqua blue
scene << Text( "center", "baseline", .2, "Back aqua blue very long string" );

// update the display box in the window using the current display list
scene << Update;
```

Figure 13.14 Rotating and Translating Text Strings

Note that the aqua blue string extends backward beyond the far clipping plane. Change the 7
to 10 in the Perspective command to see the complete string.

Using the Matrix Stack

JMP 3-D scenes use a matrix stack to keep track of the current transform. The stack is
initialized to the identity matrix, and each time a translate, rotate, or scale command is given,
the top matrix on the stack is changed.

Note: Unlike many OpenGL implementations, JMP does not use a transposed matrix.

The JSL example below uses Push Matrix and Pop Matrix to position pieces of the toy top
and then return to the origin. This is faster than using the Translate command a second time
in reverse.

Figure 13.15 Drawing With a Matrix Stack


```
toyTop = Scene Display List();
toyTop << Push Matrix;
toyTop << Translate( 0, 0, .1 );
toyTop << Color( 1, 0, 0 ); // red

// baseRadius, topRadius, height, slices, stacks
toyTop << Cylinder( 1, .2, .2, 25, 5 );
toyTop << Pop Matrix;
toyTop << Push Matrix;
toyTop << Translate( 0, 0, -.1 );
toyTop << Rotate( 180, 1, 0, 0 );
toyTop << Color( 0, 1, 0 ); // green
toyTop << Cylinder( 1, .2, .2, 25, 5 );
toyTop << Pop Matrix;

toyTop << Color( 0, 0, 1 ); // blue
toyTop << Sphere( .5, 30, 30 ); // radius, slices, stacks

toyTop << Color( 1, 1, 0 ); // yellow

// innerRadius, outerRadius, slices, rings, startAngle, sweepAngle
toyTop << Partial Disk( 1, 1.2, 25, 2, 0, 270 );

toyTop << Push Matrix;
toyTop << Translate( 0, 0, -.1 );
toyTop << Color( 1, 0, 1 ); // magenta
toyTop << Cylinder( 1, 1, .2, 25, 3 );

// baseRadius, topRadius, height, slices, stacks
toyTop << Pop Matrix;
```

```
toyTop << Push Matrix;
toyTop << Rotate( 90, 1, 0, 0 );
toyTop << Translate( 0, .5, 0 );
toyTop << Color( 0, 1, 1 ); // cyan
toyTop << Text( "center", "baseline", .2, "Toy Top" );
toyTop << Pop Matrix;

scene = Scene Box( 600, 600 ); // make a scene box...holds an OpenGL scene

New Window( "Example 3", scene ); // put the scene in a window
scene << Perspective( 45, 3, 7 );
scene << Translate( 0.0, 0.0, -4.5 );
scene << Rotate( -85, 1, 0, 0 );
scene << Rotate( 65, 0, 0, 1 );
scene << Call List( toyTop );

scene << Update; // update the display box
```

There are some cases where you want to replace the current matrix on the stack. For these cases, use the Load Matrix command.

```
Load Matrix(m)
```

where *m* is a 4x4 JMP matrix that is loaded onto the current matrix stack.

Similar is the Mult Matrix command

```
Mult Matrix(m)
```

When the Mult Matrix command is issued, the matrix on the top of the current matrix stack is multiplied by *m*.

The following matrices perform some simple commands.

Translation:

```
[1  0  0  x
 0  1  0  y
 0  0  1  z
 0  0  0  1]
```

In the following rotation matrices, c = cos(angle) and s=sin(angle).

Rotation about X axis:

```
[1  0   0  0
 0  c  -s  0
 0 -s   c  0
 0  0   0  1]
```

Rotation about Y axis*:*

```
[c  0  s  0
 0  1  0  0
-s  0  c  0
 0  0  0  1]
```

Rotation about Z axis:

```
[c -s  0  0
 s  c  0  0
 0  0  1  0
 0  0  0  1]
```

For example, here are two equivalent (except for the translation being opposite) ways to translate and rotate a display list.

```
// first way uses matrix
gl << Push Matrix;
xt = Identity( 4 ); // translate this one left by .75
xt[1, 4] = -.75;
xr = Identity( 4 ); // rotate this one, cos needs radians, not degrees
xr[2, 2] = Cos( 3.14159 * a / 180 );
xr[2, 3] = -Sin( 3.14159 * a / 180 );
xr[3, 2] = Sin( 3.14159 * a / 180 );
xr[3, 3] = Cos( 3.14159 * a / 180 );
yr = Identity( 4 );
yr[1, 1] = Cos( 3.14159 * a / 180 );
yr[1, 3] = Sin( 3.14159 * a / 180 );
yr[3, 1] = -Sin( 3.14159 * a / 180 );
yr[3, 3] = Cos( 3.14159 * a / 180 );
zr = Identity( 4 );
zr[1, 1] = Cos( 3.14159 * a / 180 );
zr[1, 2] = -Sin( 3.14159 * a / 180 );
zr[2, 1] = Sin( 3.14159 * a / 180 );
zr[2, 2] = Cos( 3.14159 * a / 180 );

// order of multiplication matters with matrices
gl << Mult Matrix( xt * xr * yr * zr );
gl << Arcball( dl, 1 );
gl << Pop Matrix;

// second way uses functions
gl << Push Matrix;
gl << Translate( .75, 0, 0 ); // translate this one right by .75
gl << Rotate( a, 1, 0, 0 ); // rotate this one in degrees
gl << Rotate( a, 0, 1, 0 ); // order of operations also matters here
gl << Rotate( a, 0, 0, 1 );
```

```
gl << Arcball( dl, 1 );
gl << Pop Matrix;
```

It is not possible to read back the current transform matrix, because the matrix exists only while the display list is drawing, not while your JSL script is creating it. If you must know its content, create it in JSL and use Load Matrix to put it on the stack.

Lighting and Normals

The following methods enable you to add lighting, materials, and normal vectors to your shapes. Using these methods, models can appear shiny or light-absorbing.

- "Creating Light Sources"
- "Lighting Models"
- "Normal Vectors"
- "Shading Model"
- "Material Properties"
- "Alpha Blending"
- "Fog"
- "Example"

Creating Light Sources

Light sources are specifications of a color, position, and direction. JSL allows for up to eight lights (numbered 0 to 7) defined by the Light command, where *n* is the number of the light.

```
Light( n, argument, value, ... value )
```

Note: To turn each light on, issue an Enable(*Lighting*) and an Enable(light*n*) command, where *n* is the light number. Then, move the light to a position in the scene with a Light(*n*, *POSITION*, *x*, *y*, *z*) command.

The value of *argument* can be any one of those shown in Table 13.3. The table shows default values for each argument.

Table 13.3 Light Arguments and Default Values

Argument	Default Value	Meaning
AMBIENT	(0, 0, 0, 1)	Ambient RGBA intensity
DIFFUSE	(1, 1, 1, 1)	diffuse RGBA intensity
SPECULAR	(1, 1, 1, 1)	specular RGBA intensity
POSITION	(0, 0, 1, 0)	(x, y, z, w) position
SPOT_DIRECTION	(0, 0, -1)	(x, y, z) direction of spotlight
SPOT_EXPONENT	0	spotlight exponent
SPOT_CUTOFF	180	spotlight cutoff angle
CONSTANT_ATTENUATION	1	constant attenuation factor
LINEAR_ATTENUATION	0	linear attenuation factor
QUADRATIC_ATTENUATION	0	quadratic attenuation factor

Note: The default values for DIFFUSE and SPECULAR in this table apply only to Light 0. For other lights, the default value is (0, 0, 0, 1) for both arguments.

The first three arguments (AMBIENT, DIFFUSE, and SPECULAR) are used to color the light. DIFFUSE is the argument that is most closely associated with the physical color of the light. AMBIENT refers to the property of the light when it functions as a background light. SPECULAR alters the way a light is reflected off a surface.

Specify the position of the light using the POSITION argument. Nonzero values of the fourth (*w*) coordinate position the light in homogenous object coordinates.
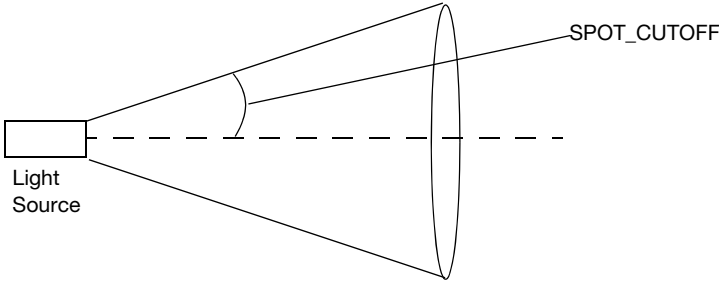
Light in the real-world decreases in intensity as distance from the light increases. Since a directional light is infinitely far away, it does not make sense to attenuate its intensity as a function of distance. However, JSL attenuates a light source by multiplying the contribution of the source by an attenuation factor

$$\text{attenuation factor } = \frac{1}{c + ld + qd^2}$$

where c = CONSTANT_ATTENUATION, l = LINEAR_ATTENUATION, and q = QUADRATIC_ATTENUATION.

To create a spotlight, limit the shape of the light to a cone. Use the SPOT_CUTOFF argument to define the side of the cone, as shown in the following illustration.

Figure 13.16 Spotlight

In addition to the cutoff angle, you can control the intensity and direction of the light distribution in the cone. SPOT_DIRECTION specifies the direction for the spotlight to point; SPOT_EXPONENT influences how concentrated the light is.

Lighting Models

Lighting models are specified with the Light Model command.

    Light Model(argument, value,...,value)

Light models specify three attributes of lights.

- The global ambient light intensity
- Whether the viewpoint is local or is an infinite distance away
- Whether lighting calculations should be performed differently for the front and back faces of objects.

Table 13.3 on page 753 shows the three valid arguments for the Light Model command.

Table 13.4 Light Model Arguments and Default Values

Argument	Default Value	Meaning
LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1)	Ambient RGBA intensity of the entire scene
LIGHT_MODEL_LOCAL_VIEWER	0 (false)	how specular reflection angles are computed
LIGHT_MODEL_TWO_SIDE	0 (false)	nonzero values imply two-sided lighting

Normal Vectors

Normal vectors point in a direction perpendicular to a surface. For a plane, all normals are the same. For a more complicated surface, normals are more complicated. JSL enables you to specify the normal vector for each vertex. These normals specify the orientation of the surface in space, necessary for lighting calculations. Accurate normals assure accurate lighting.

The normal vector is of length 1 and is perpendicular to the vertex. Typically, a vertex is shared between several polygons. A smooth shaded effect is desired, so the perpendicular at the vertex is calculated as a (possibly weighted) average of the polygon's normals. It is important to calculate the "outward" normal for polygons unless two-sided shading is enabled because only the outer face of the polygon is illuminated. With a scaled polygon, the normal's length is not 1 after scaling, and the lighting is wrong.

Normal vectors are set at the same time the surface is constructed, and are specified with the `Normal` command. Use the `Enable(NORMALIZE)` command to have the normals re-normalized to 1 each time the scene is drawn.

Shading Model

The shading model of a polygon is set using the `Shade Model` command.

    Shade Model (*mode*)

where *mode* can be SMOOTH (the default) or FLAT. SMOOTH shading interpolates the colors of the primitive from one vertex to the next. FLAT mode duplicates the color of one vertex across the entire primitive.

The following script changes the color at each of a triangle's vertices. The FILL shade model interpolates the color of the interior automatically.
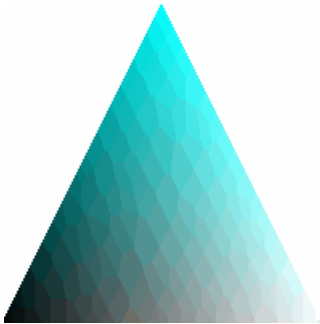
```
scene = Scene Box( 200, 200 ); // make a scene box...holds an OpenGL scene.

New Window( "Shade Model", scene ); // put the scene in a window
scene << Clear;
scene << Ortho2D( -1, 1, -1, 1 );

scene << Shade Model( SMOOTH );
scene << Polygon Mode( FRONT_AND_BACK, FILL );
scene << Begin( TRIANGLES );
scene << Color( 0, 0, 0 ); // black
scene << Vertex( -1, -1, 0 );
scene << Color( 0, 1, 1 ); // cyan
scene << Vertex( 0, 1, 0 );
scene << Color( 1, 1, 1 ); // white
scene << Vertex( 1, -1, 0 );
scene << End();
```

```
scene << Update;
```

Figure 13.17 Shading

Material Properties

To set the material properties of a surface, use the Material command.

```
Material( face, argument, value,...value )
```

face can be Front, Back, or Front_and_back. (Note that the material properties can be set separately for the front and back faces of a polygon.)

Table 13.5 shows the arguments and default values for Material arguments.

Table 13.5 Material Arguments and Default Values

Argument	Default Value	Meaning
AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color of material
DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse color of material
AMBIENT_AND_DIFFUSE		Both AMBIENT and DIFFUSE
SPECULAR	(0.0, 0.0, 0.0, 1.0)	Specular color of material
SHININESS	0	Specular exponent that can range from 0 to 128.
EMISSION	(0, 0, 0, 1)	Emissive color of material

Alpha Blending

The `BlendFunc` command allows for alpha blending. To use it, send a `BlendFunc` message to a scene, for example:

```
scene << BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
```

`SRC_ALPHA` and `ONE_MINUS_SRC_ALPHA` are OpenGL constants that tell `BlendFunc` to use alpha to blend against the existing display buffer. Disabling *z*-buffer testing or rendering primitives from back to front might be needed for some applications. By default, the *z*-buffer tests prevent anything from drawing behind a transparent polygon after it is drawn.

More information about all the constants available to `BlendFunc` (many of which are not useful to the JSL programmer) are available in the OpenGL documentation at opengl.org.

Fog

Fog enables figures to fade into the distance, making for more realistic models. All types of geometric figures can be fogged. To turn fog on, enable the `FOG` argument.

```
Enable (FOG)
```

Example

The following example uses several of the concepts presented in this section, including lighting, fog, and normalization. It draws a spinning cylinder that is affected by two lights.

```
scene = Scene Box( 300, 300 ); // make a scene box
New Window( "Cylinder", scene ); // put the scene in a window

For( i = 1, i < 360, i++,
  scene << Clear;

  // the lens is 45 degrees, near is 3 units from the camera, far is 7.
  scene << Perspective( 50, 1, 10 );

  // move the world so that 0,0,0 is visible in the camera
  scene << Translate( 0.0, 0.0, -2 );
  scene << Enable( Lighting );
  scene << Enable( Light0 );
  scene << Enable( Light1 );
  scene << Light( Light0, POSITION, 1, 1, 1, 1 ); // near viewer
  scene << Light( Light0, DIFFUSE, 1, 0, 0, 1 ); // red light
  scene << Light( Light1, POSITION, -1, -1, -1, 1 ); // behind object
  scene << Light( Light1, DIFFUSE, .5, .5, 1, 1 ); // blue-gray light
  scene << Enable( Fog );
```
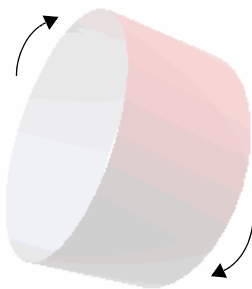
```
    scene << Enable( NORMALIZE );

    scene << Rotate( i, 1, 0, 0 );
    scene << Rotate( i * 3, 0, 1, 0 );
    scene << Rotate( i * 3 / 2, 0, 0, 1 );
    scene << Cylinder( 0.5, 0.5, 0.5, 40, 10 );
    scene << Update;
    Wait( 0.01 );
);
```
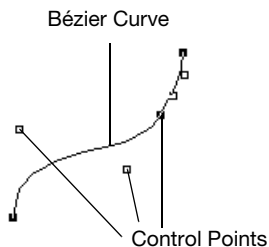
Figure 13.18 Fog

Bézier Curves

A complete discussion of Bézier curves is beyond the scope of the *Scripting Guide*. JSL has
several commands for defining and drawing curves and their associated meshes.

Figure 13.19 Bézier Curve

One-Dimensional Evaluators

To define a one-dimensional map, use the Map1 command.

```
Map1( target, u1, u2, stride, order, matrix )
```

The *target* argument defines what the control points represent. Values of the *target* argument are shown in Table 13.6. Note that you must use the Enable command to enable the argument.

Table 13.6 Map1 *Target* Arguments and Default Values

target **Argument**	**Meaning**
MAP1_VERTEX_3	(x, y, z) vertex coordinates
MAP1_VERTEX_4	(x, y, z, w) vertex coordinates
MAP1_INDEX	color index
MAP1_COLOR_4	R, G, B, A
MAP1_NORMAL	normal coordinates
MAP1_TEXTURE_COORD_1	s texture coordinates
MAP1_TEXTURE_COORD_2	s, t texture coordinates
MAP1_TEXTURE_COORD_3	s, t, r texture coordinates
MAP1_TEXTURE_COORD_4	s, t, r, q texture coordinates

The second two arguments (*u1* and *u2*) define the range for the map. The *stride* value is the number of values in each block of storage. In other words, the value is the offset between the beginning of one control point and the beginning of the next control point. The *order* should equal the degree of the curve plus one. The *matrix* holds the control points.

For example, Map1(MAP1_VERTEX_3, 0, 1, 3, 4, *<4x3 matrix>*) is typical for setting the two end points and two control points to define a Bézier line.

You use the MapGrid1 and EvalMesh1 commands to define and apply an evenly spaced mesh.

```
MapGrid1( un, u1, u2 )
```

sets up the mesh with *un* divisions spanning the range *u1* to *u2*. Code is simplified by using the range 0 to 1.

```
EvalMesh1( mode, i1, i2 )
```

actually generates the mesh from *i1* to *i2*. The *mode* can be either POINT or LINE. The
EvalMesh1 command makes its own Begin and End clause.

The following example script demonstrates a one-dimensional outlier. A random set of
control points draws a smooth curve. Only the first and last points are on the curve. Using
NPOINTS=4 results in a cubic Bézier spline.

```
boxwide = 500;
boxhigh = 400;

gridsize = 100; // bigger for finer divisions

/* We suggest you use only values between 2 and 8 (inclusively). Numbers
   beyond these might be interpreted differently, depending on implementation.
   This value is the degree+1 of the fitted curve */
NPOINTS = 4;

point = J( NPOINTS, 3, 0 );

// create an array of x,y,z triples
For( x = 1, x <= NPOINTS, x++,
   point[x, 1] = (x - 1) / (NPOINTS - 1) - .5;
      // x from -.5 to +.5
   point[x, 2] = Random Uniform() - .5; // y is random in same range

   // z is always zero, which causes the curve to stay in a plane
   point[x, 3] = 0;
);

spline = Scene Box( boxwide, boxhigh );

// data from -.5 to .5 in x and y; this is a little larger
spline << Ortho( -.6, .6, -.6, .6, -2, 2 );

spline << Enable( MAP1_VERTEX_3 );
spline << MapGrid1( gridsize, 0, 1 );
spline << Color( .2, .2, 1 ); // blue curve

spline << Map1( MAP1_VERTEX_3, 0, 1, 3, NPOINTS, point );
spline << Line Width( 2 ); // not-so-skinny curve
spline << EvalMesh1( LINE, 0, gridsize ); // also try LINE, POINT

spline << Color( .2, 1, .2 );
spline << Point Size( 4 ); // big fat green points

// show the points and label them
For( i = 1, i <= NPOINTS, i++,
```

```
        spline << Begin( "POINTS" );
        spline << Vertex( point[i, 1], point[i, 2], point[i, 3] );
        spline << End;
        spline << Push Matrix;
        spline << Translate( point[i, 1], point[i, 2], point[i, 3] );
        spline << Text( center, bottom, .05, Char( i ) );
        spline << Pop Matrix;
    );

    New Window( "Spline", spline );
```

https://www.tinaja.com/glib/bezconn.pdf offers an explanation of connecting cubic segments so that both the slope and the rate of change match at the connection point. This example does not illustrate doing so; there is only one segment here.

Two-Dimensional Evaluators

Two-dimensional evaluators follow their one dimensional counterparts, and are used in a similar way.

```
    Map2(target, u1, u2, ustride, uorder, v1, v2, vstride, vorder, matrix)
    Eval Coord2(u, v)
```

Values for the *target* argument are the same as those shown in Table 13.6 on page 759 with *Map1* replaced with *Map2* appropriately. The *u1, u2, v1,* and *v2* values specify the range of the two-dimensional mesh.

For example, Map2(MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, *<16x3 matrix>*) is typical for setting the 16 points that define a Bézier surface.

Use the MapGrid2 and EvalMesh2 commands to define and apply an evenly spaced mesh.

```
    MapGrid2(un, u1, u2, vn, v1, v2)
```

sets up the mesh with *un* and *vn* divisions spanning the range *u1* to *u2* and *v1* to *v2*. Code is simplified by using ranges that span 0 to 1.

```
    EvalMesh2(mode, i1, i2, j1, j2)
```

actually generates the mesh from *i1* to *i2* and *j1* to *j2*. The *mode* can be POINT, LINE, or FILL. The EvalMesh2 command makes its own Begin and End clause.

Using the Mouse

Mouse activity is supported through two feedback functions. The Patch Editor.jsl sample script uses these functions to support the dragging and dropping of points. Part of that script, the call-back function for mouse activity, is explained below. To run the script, open PatchEditor.jsl in the Scene3D folder inside the JMP Sample Scripts folder.

```
topClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 1, 2 );
   1;
);
frontClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 1, 3 );
   1;
);
rightClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 2, 3 );
   1;
);

Click3d = Function( {x, y, m, k, hitlist},
   If( m == 1,
      If( N Items( hitlist ) > 0,
         CurrentPoint = hitlist[1][3], /* first matrix in the list is the
   closest; 3rd element of matrix is ID*/
         CurrentPoint = 0
      );
      makePatch();
   );
   0; /* cares only about initial mouse down. return 1 if drag, release is
   needed, but then arcball does not happen. */
);

/* after one of the 3 Click2d functions figures out which axis of the model is
   represented by the screen X, Y, pass in to this common code */
dragfunc = Function( {x, y, m, ix, iy}, /* ix, iy are the index of the X, Y,
   or Z part of the coord in the points matrix */
   If( CurrentPoint > 0,
      points[CurrentPoint, ix] = (x / boxwide) * (orthoright - ortholeft) +
   ortholeft;
      points[CurrentPoint, iy] = (y / boxhigh) * (orthotop - orthobottom) +
   orthobottom;
      makepatch();
   )
);
```

When a 2-D function is called, the arguments are X, Y, M, K.

- X and Y are the coordinates of the mouse.

- M shows the state of the mouse and button. M=0 says that the mouse button is up. M=1 says that the button was just pressed. M=2 says that the button is down and the mouse is moving. M=3 says that the button was just released.

- K is related to the keys Shift, Alt, and Ctrl. K=1 for the Shift key, K=2 for the Ctrl (Command) key, and K=3 for the Alt (Option) key.

The 3-D function is called similarly. The arguments are X, Y, M, K, `hitlist`, where `hitlist` is a list of matrices

```
[znear, zfar, id1, id2, id3, ...]
```

`znear`, `zfar` is the Z distance from the camera of the near and far edge of the object. The matrices are sorted from near to far by the midpoint of `znear`, `zfar`. The `id`s in the list are the `pushname`, `loadname`, and `popname` values you just put in the display list.

The drag functions use a return value to tell if mouse processing should continue. That is the trailing "1" you see in the functions. Anything else stops the mouse tracking. This is needed because the 2-D and 3-D functions do not run in parallel. You might want the 2-D to return 0 and the 3-D to return 1 so the tracking would happen in 3-D rather than 2-D.

Pick Commands

This `SceneBox` callback gets the 2-D mouse coordinates, and then uses `Pick` to determine the "named" object under the mouse. For example, `hitlist` is a 5x5 pixel pick box around `x,y`; up to 1000 items returned, but just the leaf names. The format of the return is determined by the last argument (1 returns a simple array, 0 returns a sorted (by depth) list of arrays).

```
Track2d=function({x, y, m, k},
hitlist = theSceneBox<<pick( x, y, 5, 5, 1000, 1 );
  if ( nrow(hitlist) > 0, // something IS in the pick box
     ... hitlist[1..n] // are names that you put in the display list
  ) );
```

Contrast this with a call back `Track3d` function, where the pick rectangle is always 1x1 and picking only happens when the mouse moves. This is almost always what you want, but points are difficult to pick because the 1x1 pick area is the same small size as the point. This function lets you pick without a mouse move.

The `Track3d` function always provides a depth-sorted list of arrays; each array can describe multiple names in a hierarchy (pushname and popname construct a hierarchy of objects). The sorting can be very slow when thousands of objects are selected. The final argument (1, above) controls whether the pick function replaces the sorted list of arrays with a simple array. The simple array contains only the "leaf" names, not higher level names.

Three-Dimensional Arguments

Arguments enable you to specify special modes and settings. To enable an argument, use the
`Enable(argument)` command. To disable an argument, use the `Disable(argument)`
command. Available arguments are shown in Table 13.7.

Table 13.7 Arguments

ALPHA_TEST	LIGHT5	MAP2_TEXTURE_COORD_3
AUTO_NORMAL	LIGHT6	MAP2_TEXTURE_COORD_4
BLEND	LIGHT7	MAP2_VERTEX_3
CLIP_PLANE0	LIGHTING	MAP2_VERTEX_4
CLIP_PLANE1	LINE_SMOOTH	NORMALIZE
CLIP_PLANE2	LINE_STIPPLE	POINT_SMOOTH
CLIP_PLANE3	MAP1_COLOR_4	POLYGON_OFFSET_FILL
CLIP_PLANE4	MAP1_INDEX	POLYGON_OFFSET_LINE
CLIP_PLANE5	MAP1_NORMAL	POLYGON_OFFSET_POINT
COLOR_LOGIC_OP	MAP1_TEXTURE_COORD_1	POLYGON_SMOOTH
COLOR_MATERIAL	MAP1_TEXTURE_COORD_2	POLYGON_STIPPLE
CULL_FACE	MAP1_TEXTURE_COORD_3	SCISSOR_TEST
DEPTH_TEST	MAP1_TEXTURE_COORD_4	STENCIL_TEST
DITHER	MAP1_VERTEX_3	
FOG	MAP1_VERTEX_4	
LIGHT0	MAP2_COLOR_4	
LIGHT1	MAP2_INDEX	
LIGHT2	MAP2_NORMAL	
LIGHT3	MAP2_TEXTURE_COORD_1	
LIGHT4	MAP2_TEXTURE_COORD_2	

Improve the Display of Graphics

Hardware acceleration determines how fast and smoothly graphics are drawn. On Windows,
hardware acceleration is turned off by default.

Note: JMP uses OpenGL 1.1 to perform 3-D rendering.

If you need OpenGL and have the hardware support, turn on hardware acceleration in one of
the following ways:

• Right-click the graph and select **Use Hardware Acceleration**.

• Send the `Use Hardware Acceleration(1)` message to the scene box.

- In your script, set the global variable `Scene3DHardwareAcceleration = 1`; If you want hardware acceleration to be the default across runs of JMP, add it to your jmpStartAdmin.jsl file. Because it is a global variable, it will remain in effect until changed.

Extending JMP

External Data Sources, Analytical Tools, and Automation

This chapter discusses scripting features that are particularly useful for production settings, such as the following:

- a data feed for capturing real-time data from a laboratory instrument

- using SAS, MATLAB, R, or Python through JMP Scripting Language (JSL)

- using JMP with Microsoft Excel

- connecting to databases

- controlling JMP externally by OLE automation

- parsing XML

- publishing to JMP Public or JMP Live

- communicating with REST web services

- communicating with web APIs

Some general JSL commands that might be of particular interest for use in a production setting include Caption, Speak, Print, Write, and Mail. These commands are described in "Functions that Communicate with Users" on page 316 in the "Programming Methods" chapter.
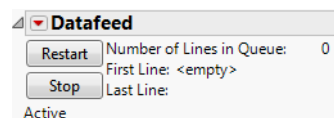
Contents

Real-Time Data Capture on Windows

A data feed is a real-time method to read data continuously, such as from a laboratory measurement device connected to a serial port. A data feed object sets up a concurrent thread with a queue for input data lines that arrive in real time and are processed with background events. You set up scripts to interpret the data lines and push the data to data tables, or do whatever else your process requires.

For example, submit this to get records from com1: and list them in the log.

```
feed = Open Datafeed(
   Connect( Port( "com1:" ), Baud( 9600 ), DataBits( 7 ) ),
   Set Script( Print( feed << GetLine ) )
);
```

Figure 14.1 A Datafeed Window Shows the Status and Offers Controls

Create a Data Feed Object

To create a data feed object, use the `Open Datafeed()` function with arguments specifying details about the connection:

```
feed = Open Datafeed( options );
```

No arguments are required. You can simply create the data feed object and then send messages to it. Messages might include connecting to a port or setting up a script to process the data coming in. However, you would typically set up the basic operation of the data feed in the `Open Datafeed()` function and subsequently send messages as needed to manage the data feed. Any of the options below work both as options inside `Open Datafeed()` or as messages sent to a data feed object.

It's a good idea to store a reference to the object in a global variable, such as *feed* above, so that you can easily send messages to the object. You can store a reference to an existing object by using a subscript; for example, to store a reference to the second data feed created:

```
feed2 = Datafeed[2];
```

Data Feed Options

To connect to a live data source, use Connect( ) and specify details for the port. Each setting takes only one argument; in this syntax summary the symbol | between argument choices means "or." The Port specification is needed if you want to connect, otherwise the object still works, but is not connected to a data feed. The last three items, DTR_DSR, RTS_CTS, and XON_XOFF, are Boolean to specify which control characters are sent back and forth to indicate when the data feed is ready to get data. Typically, you would turn on at most one of them.

```
feed = Open Datafeed(
  Connect(
    Port( "com1:" | "com2:" | "lpt1:" | ... ),
    Baud( 9600 | 4800 | ... ),
    Data Bits( 8 | 7 ),
    Parity( None | Odd | Even ),
    Stop Bits( 1 | 0 | 2 ),
    DTR_DSR( 0 | 1 ), // Data Terminal Ready
    RTS_CTS( 0 | 1 ), // Request To Send | Clear To Send
    XON_XOFF( 1 | 0 ) // Transmitter On | Transmitter Off
  )
);
```

This command creates a scriptable data feed object and stores a reference to it in the global variable *feed*. The Connect() argument starts up a thread to watch a communications port and collect lines. The thread collects characters until it has a line, and then appends it to the line queue and schedules an event to call the script.

Note: A line of data from the feed might generate multiple rows in a data table or might be a fraction of one row. You provide JSL to parse the data lines and to add rows to a data table as needed.

Set Script() attaches a script to the data feed object. This script is triggered by the Open Datafeed message whenever a line of data arrives. The argument for Set Script() is simply the script to run, or a global containing a script.

```
feed = Open Datafeed( Set Script( myScript ) );
feed = Open Datafeed( Set Script( Print( feed << Get Line ) ) );
```

A data feed script typically uses Get Line to get a copy of one line and then does something with that line. Often it parses the line for data and adds it to some data table.

Read in Real-Time Data

The term *live data feed* describes the way an external data source sends information via a physical or a logical communication link to another device. You can connect JMP to a live data feed through the serial port of your Windows computer to read a stream of incoming data in real time. Remember the following:

- The data feed must come through a standard nine-pin serial port. Data cannot be read through a USB port unless there is a driver that can simulate a serial port.

- You need to know the exact baud rate, parity, stop bits, and data bits for the attached device.

Once you obtain the numbers for your device, enter them into the Open Datafeed() command in the script below. (The 4800, even, 2, and 7 in the script below are examples, so replace them with your information). Then connect the data feed to your computer and open and run the script:

```
streamScript = Expr(
   line = feed << Get Line;
   Show( line );
   len = Length( line );
   Show( len );
   If( Length( line ) >= 1,
      Show( "Hi" );
      Show( line );
      field = Substr( line, 5, 8 );
      Show( field );
      x = Num( field );
      Show( x );
      If( !Is Missing( x ),
         dt << Add Row( {:Column1 = x} );
         Show( x );
      );
   );
);
feed = Open Datafeed(
   Baud Rate( 4800 ),
   parity( even ),
   Stop bits( 2 ),
   Data bits( 7 )
);
feed << Set Script( streamScript );
feed << Connect;
```

To ensure harmony between the communications settings for JMP and the instrument reading data from an external source, select **File > Preferences > Communications**. Refer to the documentation for your instrument to find the appropriate settings.

Manage a Data Feed with Messages

A data feed object responds to several messages, including `Connect` and `Set Script`. These are detailed above as arguments for `Open Datafeed()`. They can also be sent as messages to a data feed object that already exists:

```
feed << Connect( port( "com1:" ), baud( 4800 ), databits( 7 ), parity( odd ),
    stopbits( 2 ) );
feed << Set Script( myScript );
```

The following messages could also be used as arguments to `On Datafeed()`. However, it would be more common to send them as messages to a data feed object that is already present.

You can send lines to a data feed from a script. This is a quick way to test a data feed. If you do not have a device available, `Queue Line` does not send data to a device; it simulates getting data from a device. Include a text argument or a global that stores text:

```
feed << Queue Line( "14" );
feed << Queue Line( myValue );
```

Here is a test script to queue five lines of data:

```
feed << Queue Line( "11" );
feed << Queue Line( "22" );
feed << Queue Line( "33" );
feed << Queue Line( "44" );
feed << Queue Line( "55" );
```

Figure 14.2 Datafeed: 5 Lines Queued

To get the first line currently waiting in the queue, use a `Get Line` (singular) message. When you get a line, it is removed from the queue. Five lines were queued with the test script above, and `Get Line` returns the first line and removes it from the queue:

```
feed << Get Line
    "11"
```

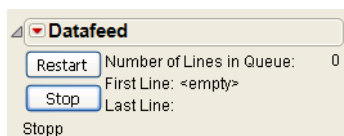**Figure 14.3** Datafeed: 4 Lines Queued

To empty all lines from the queue into a list, use `Get Lines` (plural). This returns the next four lines from the test script in list { } format.

```
myList = feed << GetLines;
    { "22", "33", "44", "55" }
```

Figure 14.4 Datafeed: 0 Lines Queued

To stop and later restart the processing of queued lines, either click the **Stop** and **Restart** buttons in the Datafeed window, or send the equivalent messages:

```
feed << Stop;
feed << Restart;
```

To close the data feed and its window:

```
feed << Close;
```

To disconnect from the live data source:

```
feed << Disconnect
```

Table 14.1 Datafeed Messages

Message	Syntax	Explanation
Open Datafeed Datafeed	feed = Open Datafeed(commands)	Creates a data feed object. Any of the following can be used as commands inside Open Datafeed or sent as messages to an existing data feed object. Datafeed is a synonym.
Set Script	feed << Set Script(script)	Assigns the *script* that is run each time a line of data is received.
Get Line	feed << Get Line	Returns and removes one line from the data feed queue.

Table 14.1 Datafeed Messages *(Continued)*

Message	Syntax	Explanation
Get Lines	feed << Get Lines	Returns as a list and removes all lines from the data feed queue.
Queue Line	feed << Queue Line(*string*)	Sends one line to the end of the data feed queue.
Stop	feed << Stop	Stops processing queued lines.
Restart	feed << Restart	Restarts processing queued lines.
Close	feed << Close	Closes the data feed object and its window.
Connect	feed << Connect(Port("com1:" | "lpt1:" | ...), Baud(9600 | 4800 | ...), Data Bits(8 | 7), Parity(None | Odd | Even), Stop Bits(1 | 0 | 2), DTR_DSR(0 | 1), RTS_CTS(0 | 1), XON_XOFF(1 | 0))	(Windows only) Sets up port settings for the connection to the device. The symbol | between arguments represents "or"; choose one argument for each setting.
Disconnect	feed << Disconnect	(Windows only) Disconnects the device from the data feed queue but leaves the data feed object active.

Examples of Data Feed

Reading Data

Here is a typical data feed script. It expects to find a string 3 characters long starting in column 11. If it does, it uses it as a number and then adds a row to the data table in the column "thickness."

```
feed = Open Datafeed( );
myScript = Expr(
    line = feed << Get Line;
    If( Length( line ) >= 14,
        x = Num( Substr( line, 11, 3 ) );
```

```
      If( !Is Missing( x ),
         dt << Add Row( {thickness = x} )
      );
   );
);
```

Assign the script to the data feed object by using Set Script:

```
feed << Set Script( myScript );
```

Set Up a Live Control Chart

Here is a sample script that sets up a new data table and starts a control chart based on the data feed.

```
dt = New Table( "Gap Width" ); // make a data table with one column
dc = dt << New Column( "gap", Numeric, Best );

// set up control chart properties
dc << Set Property(
  "Control Limits",
  {XBar( Avg( 20 ), LCL( 19.8 ), UCL( 20.2 ) )}
);
dc << Set Property( "Sigma", 0.1 );

// make the data feed
feed = Open Datafeed();
feedScript = Expr(
  line = feed << Get Line;
  z = Num( line );
  Show( line, z ); // if logging or debugging
  If( !Is Missing( z ),
     dt << Add Row( {:gap = z} )
  );
);
feed << Set Script( feedScript );

// start the control chart
Control Chart Builder(
      Show Capability( 0 ),
      Variables( Y( :gap ) ),
      Set Subgroup Size( 5 )
);
```

```
/* Either start the feed from the device or test-feed some data
to see it work (comment out one of the lines):
feed << Connect( Port( "com1:" ), Baud( 9600 ) );*/
For( i = 1, i <= 20, i++,
  feed << Queue Line( Char( 20 + Random Uniform() * .1 ) );
  Wait( .1 ); // simulate a delay in the feed
);
```

Store the Script in a Data Table

You can further automate the production setting by placing a data feed script such as the one above in an `On Open` data table property. A property with this name is run automatically each time the table is opened (unless you set a preference to suppress execution). If you save such a data table as a template, opening the template runs the data feed script and creates a new data table.

Dynamic Link Libraries (DLLs)

Note: 64-bit JMP cannot load 32-bit DLLs. You must recompile a 32-bit DLL for JMP to be able to load it.

You can extend JMP functionality by using JMP Scripting Language (JSL) to load a DLL and call functions exported by that DLL. There is one JSL command and six messages that implement this functionality.

```
dll_obj = Load DLL("path" <, AutoDeclare(Boolean | Quiet | Verbose) |Quiet |
    Verbose > )
```

`Load DLL()` loads the DLL in the specified *path*. Use the `AutoDeclare(Quiet)` argument to suppress log window messaging.

Use the `Declare Function` message to declare a function that is defined in the DLL.After you declare the function, you can call it.

```
dll_obj <<Declare Function("name", Convention(named_argument),
    Alias("string"), Arg(type, "string"), Returns(type), other_named_arguments)
```

The `Alias` defines an alternate name that you can use in JSL. For example, if you declared `Alias("MsgBox")` for a function that is named `"Message Box"` in the DLL, then you would call it as follows:

```
result = dll_obj <<MsgBox(...)
```

Here are the named arguments for `Convention`:

• STDCALL or PASCAL

- CDECL

The type argument for both `Arg` and `Returns` can be one of the following:

Table 14.2 Types for Arg and Returns

Int8	UInt8	Int16	UInt16
Int32	UInt32	Int64	UInt64
Float	Double	AnsiString	UnicodeString
Struct	IntPtr	UIntPtr	ObjPtr

See the *JSL Syntax Reference* for the `Declare Function` message arguments.

Finally, use the `UnLoadDLL` message to unload the DLL:

```
dll_obj << UnLoadDLL
```

Note: Refer to the documentation for that function provided by the DLL author when you declare a function. If the argument types and calling convention declared do not match the actual function in the DLL, calling the function could cause JMP to terminate.

Example

In the following example, the DLL contains a function named `netPresentValue`, which takes four parameters. `Declare Function()` defines these parameters and the data type that the function returns so that JMP can successfully call the function.

```
xftdll = Load Dll( "c:\ExtFunctionTests\ExtFunctionTests_x64.dll" ),

Show( xftdll );

xftdll << Declare Function(
  "netPresentValue",
 Convention( CDECL ),
  Alias( "npv" ), // use this name to call the function in JMP
  Arg( Double, "discount rate" ),
  Arg( Double, "cash flows per period" ),
  Arg( Int32, "number of cash flows" ),
  Arg( Double, Array, "array of cash flow values" ),
  Returns( Double )
);

result = xftdll << npv( 0.10, 1, 10, [ 100, 100, 100, 100, 100,
```

```
  100, 100, 100, 100, 100 ]
);
```

Other DLL Messages

The Show Functions message sends any functions that have been declared using Declare
Function to the log:

```
dll_obj << Show Functions;
```

If you are writing your own DLL, you can create functions in it using JSL. The `Get
Declaration` JSL message sends any JSL functions in your DLL to the log:

```
dll_obj << Get Declaration JSL;
```

Use Sockets in JSL

Another tool that can be useful in establishing a live data feed is JSL Sockets. You can create
two types of sockets using JSL:

Stream   Stream sockets create a reliable connection between JMP and another computer. The
other computer might be running JMP, or it might be a vending machine, data collector,
printer, or other device that is capable of socket communication. Some devices implement
their interface as an HTTP web server.

Datagram   Datagram sockets create a less reliable connection between JMP and another
computer. Datagrams are connectionless and information might arrive multiple times, not
at all, and out of order. A datagram connection does not include all the overhead of a
stream connection that does guarantee reliability. Because datagrams are connectionless,
the destination address must be supplied each time (and for the same socket that can be
different each time).

Once a socket is created, it can do two things: wait for a connection from another socket, or
make a connection to another socket. Here is a simple example program that makes a
connection to another computer's web server to get some data:

```
tCall = Socket( );
tCall << Connect( "www.jmp.com", "80" );
tCall << Send( Char To Blob( "GET / HTTP/1.0~0d~0a~0d~0a", "ASCII~HEX" ) );
tMessage = tCall << Recv( 1000 );
text = Blob To Char( tMessage[3] );
Show( text );
tCall << Close( );
```

The first line creates a socket and gives it a reference name (`tCall`). By default, a stream socket is created. You can designate the type of socket to create with an optional argument: `socket(STREAM)` or `socket(DGRAM)`.

The second line connects the `tCall` socket to port 80 (which is generally the HTTP port) of the JMP website.

The third line sends a `GET` request to the JMP web server; this message tells the JMP web server to send the JMP home page back. The / that follows the word GET should be the path to the page to be opened. A / opens the root page.

The fourth line receives up to 1000 bytes from the JMP web server and stores a list of information in `tMessage`. Each socket call returns a list. The first element of the list is the name of the call, and the second is a text message, which might be `ok` or a longer diagnostic message. Additional elements, if present, are specific to the call. In this case, the third element in the list is the data received.

The fifth line converts the binary information received into a character string. `tMessage[3]` is the third item in the list returned by `Recv`; it is the data from the JMP web server.

The sixth line displays the data in the log.

The last line closes the socket. The web server has already closed the far end, so this socket either needs reconnecting or proper disposal (`close`).

See the Scripting Index in the Help menu for more examples. The JMP Samples/Scripts folder also contains several examples of scripts using sockets.

Socket-Related Commands

Before creating and using a socket, you might need to retrieve information about the end that you want to connect to. `GetAddrInfo( )` and `GetNameInfo( )` each takes an address argument and an optional port argument and returns a list of information. For example:

```
Print( Get Addr Info( "www.sas.com" ) );
Print( Get Addr Info( "www.sas.com", "80" ) );
Print( Get Name Info( "149.173.5.120" ) );
    {"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120",
        "0"}}
    {"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120",
        "80"}}
    {"Get Name Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0",
        "chess.exnet.sas.com", "0"}}
```

Sometimes there can be more than one answer. In that case, the sublist might be repeated one or more times. These functions can be quite slow; you probably should not try to build a data table of every website name with it. For IPV6 compatibility, you should generally use names like "www.sas.com" rather than the numerical form of an address.

Messages for Sockets

Once you have created a socket with `Socket( )`, there are many messages that you can send to it.

connect   Connects to a listening socket. Returns a list: {`"connect"`, `"ok"`} if the connection was successful; or an error if not (for example, {`"connect"`, `"CONNREFUSED: The attempt to connect was forcefully rejected. "`}).

close   Closes the connection when you are finished with it. Returns a list (for example, {`"Close"`, `"ok"`}).

send   Sends a STREAM message to the other end of the socket.

sendto   Sends a DGRAM message to the other end of the socket.

recv   Receives a STREAM message. The data comes back in a list, along with some other information. `Recv` takes a required numeric argument that specifies the number of bytes to accept.

recvfrom   Receives a DGRAM message.

ioctl   Controls the socket's blocking behavior. By default, JMP sockets block if no data is available; the socket does not return control to the JSL program until data *is* available. This makes scripts easy to write, but not particularly robust if the remote end of the connection fails to supply the data. A socket that is set for non-blocking behavior always returns immediately, either with an `ok` return code and some data, or with a "WOULDBLOCK: ..." return code, which means if it were a blocking socket, it would have to wait (block progress of the next JSL statement) until data became available.

Important: Background operations that use a JSL callback avoid this issue; a socket used in a background `recv`, `recvfrom`, or `accept` is set to non-blocking and is polled during `wait` statements and when JMP is otherwise idle.

`Ioctl` returns a list. For example, {`"ioctl"`, `"ok"`}, or {`"ioctl"`, `"NOTCONN: The socket is not connected. "`} if the socket has not been bound (see `bind`, below) or connected already.

bind   Tells the server socket what address the client socket listens on. `Bind` associates a port on the local machine with the socket. This is required before a socket can `Listen`. (See below). `Bind` is not usually used on sockets that connect; the operating system selects an unused port for you. `Bind` is needed for a server because anyone that wants to connect to the server needs to know what port is being used. A common port is 80, the HTTP port. `Bind` returns a list. For example, {`"bind"`, `"ok"`}, or {`"bind"`, `"ADDRNOTAVAIL: The specified address is not available from the local machine. "`} if you try binding to a name that is not on your machine. Another socket can connect to this socket if it knows your machine name and the number.

listen   Tells the server socket to listen for connections. A listening socket is listening for connections from other sockets. You need only to put the socket into listen mode once. Accept (see below) is used to accept a connection from another socket. Listen returns a list. For example, {"listen", "ok"}, or {"listen", "INVAL: The socket is (or is not, depending on context) already bound to an address. or, Listen was not invoked prior to accept. or, Invalid host address. or, The socket has not been bound with Bind. "} if your bind call did not succeed.

accept   Tells the server socket to accept a connection and return a new connected socket. Accept returns a list stating what happened, and if successful, a new socket that is connected to the socket at the other end. For example, the returned list might be {"Accept", "ok", "localhost", socket( )}. In this case localhost is the name of the machine that just connected, and the fourth argument is a socket that you can use to send or recv a message.

getpeername   Asks about the other end of a connection. GetPeerName returns a list with information about the other end's socket: {"getpeername", "ok", "127.0.0.1", "4087"}. If this is a server socket, you can discover the address and port of the client that connected. If this is a client socket, you re-discover the name and port of the server that you used in the connect request.

getsockname   Asks about this end of a connection. GetSockName returns a list with information about this socket: {"getsockname", "ok", "localhost", "httpd"}. If this is a client socket, you can discover the port the operating system assigned. If this is a server socket, you already know this information from a bind you already made.

Database Access

Run Query Builder Queries

Query Builder is the preferred method for selecting and importing data from a SQL database without writing SQL statements. You can preview the data before importing it into a data table. Share your queries so that other users can customize and run the queries.

After you create the query, you can run the query with a JSL script. Use Include() to include the query in your script without opening it.

```
query = Include( "$DOCUMENTS/Airline.jmpquery" );
query << Run Foreground( );
```

Instead of running the query in the foreground, you can use Run Background() to run the query in the background or Run() to use the Query Builder preference for running queries. By default, queries run in the background.

See the *JSL Syntax Reference* for more information about query messages.

Open Database Function

JMP supports ODBC access to SQL databases through JSL with the Open Database function.

```
dt = Open Database(
    "Connect Dialog" | "DSN=...", // data source
    "sqlStatement" | "dataTableName" | "SQLFILE=...", // SQL statement
    Invisible,  //Optional keyword to hide the table upon importing it
    "outputTableName" // new table name
);
```

Note: The characters $# -+/%()&|;? in a database table name must be quoted with square brackets [].

The first argument is a quoted connection string to specify the data source. It should be either of the following:

- "Connect Dialog" to display the Select Data Source window (Windows) or Choose DSN window (macOS).

- "DSN=" and then the data source name and any other information needed to connect to the data source. On Windows, the data source name is shown in the name column of the User DSN or System DSN tab of the ODBC Data Source Administrator. On macOS, the DSN is shown in the ODBC Manager or iODBC Driver Manager. The rest of the string depends on the ODBC data source.

For example:

```
"DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/16/Samples/Import Data;"
```

The second argument is a double-quoted string that can be one of the following:

1. An SQL statement to execute. For example, the second argument might be a SELECT statement in a quoted string like the following:

   ```
   "SELECT AGE, SEX, WEIGHT FROM BIGCLASS"
   ```

   The SQL must conform to the SQL that the data source supports. That is, a table named "Big Class" would need to be appropriately quoted because of the space between "Big" and "Class" (if whitespace is supported). The method for quoting depends on the data source and is usually done with ", ', ', or [] characters.

2. The name of a data table. In this case, the effect is an SQL "SELECT * FROM" statement for the data table indicated. For example, Open Database would in effect execute the statement "SELECT * FROM BIGCLASS" if you specify this for the second argument:

   ```
   "BIGCLASS"
   ```

3. "SQLFILE=" and a path to a text file containing an SQL statement to be executed. For example, with the following argument, JMP would attempt to open the file mySQLFile.txt from the C:\ directory and then execute the SQL statement in the file.

```
"SQLFILE=C:\mySQLFile.txt"
```

The optional `Invisible` argument creates a hidden data table. Hidden data tables remain in memory until they are explicitly closed, reducing the amount of memory that is available to JMP. To close the hidden data table, call `Close(dt)`, where `dt` is the data table reference.

The optional *outputTableName* argument is optional and specifies the name of the output table to be created, if any. Note that `Open Database` does not always return a data table. The return value might be null. Whether it returns a data table depends on the type of SQL statement executed. For example, a SELECT statement would return a data table, but a DROP TABLE statement would not.

To save a table back to a database through JSL, send the data table reference a `Save Database( )` message:

```
dt << Save Database( "connectInfo", "TableName" );
```

The first argument works the same way as it does in `Open Database`. Note that some databases do not allow you to save a table over one that already exists. If you want to replace a table in a database, use a DROP TABLE SQL statement in an Open Database command:

```
Open Database ( "connectinfo", "DROP TABLE TableName" );
```

Note: JMP 13 or later can save tables with spaces and mixed case in the names if the data source supports them. Most data sources do support spaces, except Apache Hive and Apache Hadoop. Mixed case is preserved, but for the most part, SQL is case insensitive.

The following script opens a database with an SQL query, saves it back to the database under a new name, and then deletes the new table.

```
dt = Open Database(
   "Connect Dialog",
   "SELECT age, sex, weight FROM \!"Bigclass$\!"",
   "My Big Class"
);
dt << Save Database( "Connect Dialog", "MY_BIG_CLASS" );
Open Database( "Connect Dialog", "DROP TABLE BIGCLASS.MY_BIG_CLASS" );
```

Note: When you import data from an ODBC database, a table variable is added that can contain user ID and password information. To prevent this from happening, set the following preference: `pref(ODBC Hide Connection String(1))`. You can also select the **File** menu (Windows) or **JMP** menu (macOS), **Preferences > Tables,** and then select **ODBC Hide Connection String**. See *Using JMP*.

Creating a Database Connection and Executing SQL

You can use the following functions to handle more complex database operations:

```
db = Create Database Connection( "Connection String With Password" );
Execute SQL( db, "SQL statement", <invisible>, <"New Table Title"> );
Close Database Connection( db );
```

Using these three functions, you can open a connection, call Execute SQL several times, and then close the connection. Create Database Connection returns a handle for use in Execute SQL and Close Database Connection.

Depending on the SQL submitted, a table might or might not be returned. A SELECT statement typically returns a JMP table. INSERT INTO would not return a table, because it is modifying one in the database.

Examples

Open a connection to your database:

```
dbc = Create Database Connection(
    "DSN=dBASE Files;DBQ=$SAMPLE_IMPORT_DATA/;"
);
```

Execute one or more SQL statements using this connection:

```
dt = Execute SQL( dbc,
    "SELECT HEIGHT, WEIGHT FROM Bigclass", "NewTable"
);
```

When you are finished, close your connection.

```
Close Database Connection( dbc );
```

You might want to create a database connection without selecting a table. The script would open the Database Open window with the connection already showing. We strongly recommend that you encrypt this script. Here's an example:

```
dbc = Create Database Connection(
    "DSN=dBASE Files;DBQ=$SAMPLE_IMPORT_DATA/;UID=MyDatabase;pwd=MyPassword");
Main Menu( "Open Table" );
```

Note: Execute SQL() creates a connection string that includes the cleartext password in the resulting data table. We encourage you to create the SQL query in Query Builder and then run the query in JSL. Then the password is not included in the data table.

Write a SQL Query

The Query() function enables you to manipulate JMP data tables using SQL statements. The SQL statement is the only required argument to Query(). However, most interesting SQL statements refer to one or more data tables. Any data table that the SQL statement refers to must be passed in using a *table reference* argument.

A table reference argument can be a data table reference (dt in the following example):

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query( dt, "SELECT * FROM 'Big Class' WHERE age > 14;" );
```

The SQL statement in the preceding example includes the name of the JMP data table. If the data table path is long, you might want to use an alias for the data table in the SQL statement. To use an alias, pass the table reference as a Table() argument:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query( Table( dt, "t1" ), "SELECT * FROM t1 WHERE age > 14;" );
```

Using Table() to alias your data tables also enables you to write SQL statements for multiple queries with tables that have different names without rewriting the statements.

Private and Hidden Data Tables

If you want the data table that is produced by Query() to be either private or hidden, you can pass either Private or Invisible as arguments to Query().

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query( dt, invisible, "SELECT * FROM 'Big Class' WHERE age > 14;" );
```

Returning a Single Value

The Query() function normally returns the result of the query as a JMP data table. However, sometimes you might write a SQL query that returns a single value; you would rather have Query() return that value instead of placing the value in a JMP data table. You can do that by passing the Scalar argument.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query(
  Table( dt, "t1" ),
  Scalar,
  "SELECT AVG(height) FROM t1 WHERE age > 14;"
);
```

The preceding query returns 65, the average height of students older than 14.

Work with SAS

Make a SAS DATA Step

Sending `Make SAS Data Step` to a data table returns the text for a SAS DATA Step that can re-create the data table in SAS. For example,

```
dt << Make SAS Data Step
```

prints a DATA Step to the log that can be used in the SAS Program Editor.

Sending `Make SAS Data Step Window` produces this code in a window with a .SAS suffix, so that it can be easily sent to SAS.

Create SAS DATA Step Code for Formula Columns

Sending `Get SAS Data Step for Formula Columns` to a data table includes column formulas in the SAS data step code. Here is an example that outputs the formula for the Ratio column:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Ratio", Formula( :height / :weight ));
dt << Get SAS Data Step for Formula Columns;
```

This script output the following code to the log:

```
/*%PRODUCER: JMP - DataTable Formulas */
/*%TARGET: Ratio */
/*%INPUT: height */
/*%INPUT: weight */
/*%OUTPUT: Ratio */
/* Code to score Ratio */
Ratio =height/weight
drop ;
```

To get formulas for all columns, omit the column names as shown here:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Get SAS Data Step for Formula Columns;
```

You can also include column formulas in scoring code for SAS Model Manager. Send `Get MM SAS Data Step for Formula Columns` to the data table.

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
dt << Get MM SAS Data Step for Formula Columns;
```

The results of this script are also shown in the log.

As with `Get SAS Data Step for Formula Columns`, specifying column names is optional.

SAS Variable Names

`SAS Open For Var Names()` opens a SAS data set only to obtain the names of its variables, returning those names as a list of strings.

The rules for SAS variable names are more strict than those of JMP. The `SAS Name` function converts JMP variable names to SAS variable names, changing special characters and blanks to underscores, and various other transformations to produce a valid SAS name.

```
result = SAS Name(name);
result = SAS Name({list of names});
```

If the argument is a list of names, the result is a blank-separated character string of names. For example,

```
SAS Name({"x 1", "x 2"})
```

produces

```
" x_1 x_2"
```

Connect to a SAS Metadata Server

You can connect to a SAS server and work directly with SAS data sets. Making connections and interacting with SAS data sets is scriptable through JSL. For information about connecting to a metadata server in the JMP interface, see *Using JMP*.

Note: If you connect to a physical workspace server, there is no metadata server involved, so metadata security is never applied. You must connect to a SAS Metadata Server and then connect to a logical workspace server. Then metadata security is enforced on the metadata-defined libraries you access.

Make the Connection

First, use the `Meta Connect()` function to connect to a SAS metadata server:

```
connected = Meta Connect( "MyMetadataServer", port );
```

To specify the SAS version for the metadata server, use the `SASVersion` named argument:

```
connected = Meta Connect( "MyMetadataServer", port, SASVersion("9.4") );
```

If you supply only the machine name (for example, myserver.mycompany.com) and the port, you are prompted to provide the authentication domain, your user name, and your password. You can also specify all that in JSL:

```
connected = Meta Connect( "MyMetadataServer", port, "authdomain", "user
    name", "password" );
```

When you are finished using the SAS metadata server, use `Meta Disconnect()` to disconnect the connection. No arguments are necessary; the command closes the current metadata server connection.

You can see the repositories that are available on a metadata server and set the one that you want to use:

```
Meta Get Repositories();
    {"Foundation"}
Meta Set Repository( "Foundation" );
```

Note that if there is only one repository available, it is selected automatically and you do not need to explicitly set it.

Once your repository is set, you can view the servers that are available:

```
mylist = Meta Get Servers();
    {"SASMain", "Schroedl", "SASMain_ja", "SASMain_zh", "SASMain_ko",
        "SASMain_fr", "SASMain_de", "SASMain_Unicode"}
```

Next, set your SAS connection. You can also use this command to connect directly to a local or remote server instead of using a metadata server.

```
conn = SAS Connect( "SASMain" );
```

Now you can send `Disconnect` and `Connect` messages to the `conn` object to close and open the SAS connection.

```
conn << Disconnect();
conn << Connect();
```

This is an example of using an object and messages with SAS server connections. You might have also connected to SAS servers using global functions. If so, the `Disconnect` and `Connect` messages do not affect those global connections. However, if there is no active global connection, that global connection is set to the connection opened by the object.

Automatically Connect SAS Libraries

When you connect to a SAS server, use `Connect Libraries` to automatically connect metadata-defined libraries.

```
conn = SAS Connect( "SASMain", Connect Libraries( 1 ) );
```

All metadata-defined libraries are connected, which can slow down your connection.

To connect specific libraries later, use the `SAS Connect Libref()` function or `Connect Libref` message to a SAS server object.

View SAS Libraries

After you connect to a SAS server, use the `Get Lib Ref()` function to view the libraries on that server:

```
conn << Get Lib Refs();
    {"BOOKS", "EGSAMP", "GENOMICS", "GISMAPS", "JMPSAMP", "JMPTEST",
    "MAILLIB", "MAPS", "OR_GEN", "ORION_RE", "ORSTAR", "SASHELP",
    "SASUSER", "TEMPDATA", "TSERIES", "V6LIB", "WORK", "WRSTEMP"}
```

If the library containing the data that you want is not assigned, assign it:

```
conn << Assign Lib Refs( "MyLib", "$DOCUMENTS/data" );
```

Open SAS Data Sets

First, assign a SAS library reference:

```
conn << Assign Lib Refs( "MyLib", "$DOCUMENTS" );
```

The first argument is any name that you want to use to refer to the library reference. The second is the path on the server where the data sets are located.

Next, get the list of data sets in the selected library:

```
datasets << Get Data Sets( "MyLib" );
    {"ANDORRA", "ANDORRA2", "ANYVARNAME", "BOOKS", "BOOKSCOPYNOT", "BOOKS_VIEW",
        "CATEGORIES", "DATETIMETESTS", "MOREUGLY", "NOTTOOUGLY", "PAYPERVIEW",
        "PUBLISHERS", "PURCHASES", "PURCHASES_FULL",
```

Now you can open a data set:

```
conn << Import Data( "MyLib", "PURCHASES" );
```

or

```
conn << Import Data( librefs[1], datasets[12] );
```

or

```
conn << Import Data( "MyLib.PURCHASES" );
```

Now you can get information about any SAS data set in that library. For example, you can get a list of variables:

```
conn << Get Var Names( "MyLib.PURCHASES" );
    {"purchaseyear", "purchasemonth", "purchaseday", "bookid", "catid",
    "pubid", "price", "cost"}
```

With that information, you can choose to import only part of the data set by specifying the variables to import.

```
conn << Import Data( librefs[1], datasets[12], columns( bookvars[1],
    bookvars[2], bookvars[4] ) );
```

Note: The name specified in `Import Data` can be a friendly metadata library name or a SAS 8-character library name.

Save SAS Data Sets

To save a JMP data table or an imported SAS Data Set, use the SAS `Export Data()` command:

```
conn << Export Data( dt, librefs[1], datasets[4], ReplaceExisting );
```

Run a Stored Process

To get a reference to a stored process:

```
stp = Meta Get Stored Process( "Samples/Stored Processes/Sample: Hello World"
    );
```

There is no way to acquire a list of stored processes through JSL; you must know the path to the stored process that you want to run.

To run it, send the stored process a message:

```
stp << Run();
```

Submit SAS Code from JMP

You can also directly submit SAS code and get back SAS results. For example:

```
conn << Submit( "proc print data=sashelp.class; run;" );
```

Two optional arguments control whether you see the output and the SAS log in JMP:

```
conn << Submit( "SAS Code" <,No Output Window(True|False)> <,Get
    Log(True|False )> );
```

You can also see the SAS Log at any time using the command

```
conn << Get Log();
```

`Get Log()` returns the contents, which can be placed in a JSL variable and used like any JSL string.

Get the Values of SAS Macro Variables

JMP provides several methods for querying SAS macro variables.

To show the `systime` value:

```
systime = sas << Get Macro Var("SYSTIME");
show(systime);
```

To show the defined SAS macro variables:

```
macro_names = sas << Get Macro Var Names();
show(macro_names);
```

To iterate through the SAS macro variables and print out the values:

```
macro_names = sas << Get Macro Var Names();
For( i = 1, i <= N Items( macro_names ), i++,
    macro_value = sas << Get Macro Var( macro_names[i] );
    output = macro_names[i] || " " || Char( macro_value );
    Show( output );
);
```

To submit SAS code that defines "test" as a SAS macro variable and then gets the value from SAS:

```
sas << Submit( "%let test = 1;" );
test = sas << Get Macro Var( "test" );
Show( test );
```

All macro variable values will evaluate to numbers, if possible, otherwise they will be characters.

Preferences

To get the current SAS version preference, use:

```
Get SAS Version Preference();
```

To set the current SAS version preference, use:

```
Preference( SAS Integration Settings( SASVersion( "9.4" ) ) );
```

Sample Scripts

The JMP Samples/Scripts/SAS Integration folder contains sample scripts. To run the stored process scripts successfully, the stored processes need to be placed on your SAS Metadata Server. The stored processes can be found in the sampleStoredProcesses.spk file, also in this folder.

To import sampleStoredProcesses.spk into your SAS Metadata Server:

Caution: We recommend that you import these stored processes into a SAS Metadata Server that is used for testing rather than into a production system.

1. Run SAS Management Console.

2. Connect to your SAS Metadata Server using an account with administrative privileges.

3. Expand the BI Manager node in the left pane of the SAS Management Console.

4. Navigate to the folder in the tree under which you would like to create the imported sample stored processes.

5. Right-click that folder in either the left pane or the right pane of the SAS Management Console and select **Import**.

    The Import Wizard appears.

6. Enter the full path to sampleStoredProcesses.spk or use the **Browse** button to navigate to it.

7. Select **All Objects** in the Import Options section of the wizard.

8. Click **Next**.

    The next panel reports that during the import process, you must specify values for **Application servers** and **Source code repositories**.

9. Click **Next**.

    Select which of the application servers defined in your SAS Metadata Server that you would like to use to execute the imported stored processes.

10. Select an application server from the drop-down list under Target.

11. Click **Next**.

    Select the source code repository (directory) defined on your SAS Metadata Server where you would like the SAS code for the imported stored processes to be placed.

12. Select a source code repository from the drop-down list under Target Path.

13. Click **Next**.

    The next panel gives a summary of what occurs if you click **Import**.

14. Review the information about the panel, and if it looks correct, click **Import**.

15. During the Import process, you might be asked to provide login credentials for connecting to the metadata server to perform the import. Provide credentials with administrative privileges and click **OK**.

After the import completes, you will find a folder named BIP Tree under the folder that you imported the stored processes into. Under BIP Tree is a folder named JMP Samples, and in the JMP Samples folder are two sample stored processes: Shoe Chart and Diameter.

Please note that the paths to the sample stored processes needs to be adjusted in the sample scripts storedProcessHTML.jsl and storedProcessJSL.jsl to match the folder into which you imported the sample stored processes. Otherwise, these scripts will not work correctly.

Work with MATLAB

MATLAB, a product of MathWorks Inc., provides an interactive working environment for analyzing and visualizing computational models. MATLAB is available for Windows, Mac OS X, and Linux (64-bit). JMP supports MATLAB on both Windows and macOS platforms.

You can interact with MATLAB using JMP Scripting Language (JSL):

• Submit statements to MATLAB from within a JSL script.

• Exchange data between JMP and MATLAB.

• Display graphics produced by MATLAB.

See the *JSL Syntax Reference* for more information about using MATLAB functions in JMP.

Textual output and error messages from MATLAB appear in the log window.

Install MATLAB

MATLAB must be installed on the same computer as JMP. For the supported version of MATLAB, see the JMP website: https://www.jmp.com/system/.

How JMP Finds MATLAB

JMP delays loading MATLAB until a JSL-based script requires access to it. When you run a JSL script that calls MATLAB, JMP locates the software based on the operating system's PATH environment variable (for example, C:\Program Files\MATLAB\R2012a\).

Test Your Install

To verify that your computer is able to run JSL-based scripts using MATLAB:

1. Run the following JSL script:

   ```
   MATLAB Init();
   MATLAB Submit( "m = magic(3)" );
   magicMat = MATLAB Get( m );
   Show( magicMat );
   MATLAB Term();
   ```

   The MATLAB function M = magic(3) returns a 3-by-3 matrix using integers in the range of 1 to 3^2 with equal row and column sums. This matrix is called a *magic square*.

2. Select **View > Log**.

   You should see the following response in the log window:

   ```
   m =
   ```

```
        8      1      6
        3      5      7
        4      9      2


    magicMat =
    [ 8 1 6,
      3 5 7,
      4 9 2];
    0
```

If you see the following message in the log window:

    An installation of MATLAB cannot be found on this system.

1. Add a new environment variable with the name of MATLABROOT and a value of C:\Program
   Files\MATLAB\R2012a\ or C:\Program Files (x86)\MATLAB\R2012a\.

   Note: The path entered depends on the MATLAB installation path.

2. Verify that the MATLAB path is included in the PATH variable.

3. Run the script again to ensure JMP can access MATLAB.

Work with R

You can interact with R using JSL:

• Submit statements to R from within a JSL script.

• Exchange data between JMP and R.

• Display graphics produced by R.

Text output and error messages from R appear in the log window.

Installing R

Install R on the same computer as JMP. You can download R from the Comprehensive R
Archive Network website:

https://cran.r-project.org

Note: For macOS Catalina and later versions, download the notarized and signed package.

Because JMP is a 64-bit application, you must install the corresponding 64-bit version of R. For the supported version of R, see the system requirements on the JMP website: https://www.jmp.com/system.

Override Default R Install Location on Windows

Normally JMP determines the R_HOME environment variable internally if it is not defined in the Windows system registry:

computer\HKEY_LOCAL_MACHINE\SOFTWARE\R-code\R\InstallPath

You might have installed more than one version of R on your computer. To override the default R installation location, define the R_HOME environment variable using either of the two following methods:

1. Create the variable in your system environment variables using the Control Panel, select **Start > Control Panel > System > Advanced system settings**.

2. Click **Environment Variables**.

3. In the System variables pane, click **New**.

4. Type R_HOME for the **Variable name**.

5. Type the path to the R .exe file (for example, C:\Program Files\R\R-2.15.3).

6. Click **OK** and click **OK** again to close the System Properties window.

or

Create the variable using the JSL Set Environment Variable() function:

```
Set Environment Variable( "R_HOME", "C:\Program Files\R\R-2.15.3" );
```

Override the Default R Install Location on macOS

You might have installed more than one version of R on your computer. To use an older version, change the R_HOME environment variable to point to the R version that you want to use.

To specify the R_HOME location, follow these steps:

1. Open a terminal window.

2. Type the following code, where *value* is the location of R.

   export *R_HOME=value*

3. Press Enter.

How JMP Finds R on Windows

JMP delays loading R until a JSL-based script requires access to it. When JMP needs to load R, it follows the standard steps for finding R on a Windows computer:

1. Look up the environment variable R_HOME.

    If the variable exists, load R from the specified directory.

2. If the environment variable R_HOME does not exist, look up the InstallPath value in the Windows registry under the following key:

    HKEY_LOCAL_MACHINE\SOFTWARE\R-core\R

    If the InstallPath value exists, load R from the specified directory.

3. If the InstallPath value does not exist, an error message states that R could not be found.

How JMP Finds R on macOS

JMP uses the value of the R_HOME environment variable if the variable exists. Otherwise, it looks in /Library/Frameworks/R.framework/Versions/Current/Resources for R.

Testing Your Setup

To test that your computer is able to run JSL-based scripts that use R, run the following JSL script:

```
R Init( );
R Submit( "
  x <- 1:5
  x
" );
R Term( );
```

You should see the following output in the log:

```
[1] 1 2 3 4 5
```

JMP to R Interfaces

The following JMP interfaces are provided to access R. The basic execution model is to first initialize the R connection, perform the required R operations, and then terminate the R connection. In most cases, these functions return 0 if the R operation was successful, or an error code if it was not. If the R operation is not successful, a message is written to the log. The single exception to this is R Get(), which returns a value.

R JSL Scriptable Object Interfaces

The R interfaces are also scriptable using an R connection object. A scriptable R connection object can be obtained using the R Connect() JSL function.

Conversion Between JMP Data Types and R Data Types

Table 14.3 shows what JMP data types can be exchanged with R using the R Send( ) function. Sending lists to R recursively examines each element of the list and sends each base JMP data type. Nested lists are supported.

Table 14.3 Equivalent JMP and R Data Types for R Send( )

R Data Type	JMP Data Type
Double	Numeric
String	String
Double Matrix	Matrix
List	List
Data Frame	Data Table
Integer	Row State
Date and Time	Datetime
Time/Duration	Duration
Paired List	Associative Array

Example

```
R Init();
X = 1;
R Send( X );
S = "Report Title";
R Send( S );
M = [1 2 3, 4 5 6, 7 8 9];
R Send( M );
R Submit( "
X
S
M
" );
```

```
R Term();
```

Table 14.4 shows what JMP data types can be exchanged with R using the R Get() function. Getting lists from R recursively examines each element of the list and sends each base R data type. Nested lists are supported.

Table 14.4 Equivalent JMP and R Data Types for R Get( )

R Data Type	JMP Data Type
Double	Numeric
Logical (Boolean)	Numeric (0 | 1)
String	String
Integer	Numeric
Date and Time	Datetime
Time/Duration	Duration
Factor	Expanded to a list of Strings or a Numeric matrix
Data Frame	Data Table
List	List of converted R data types
Matrix	Numeric Matrix
Numeric Vector	Numeric Matrix
String Vector	List of Strings
Graph	Picture
Time Series	Matrix
Paired List	Associative Array

JMP Scoping Operators and R

A JMP object sent to R using R Send() uses the same JMP reference as the name of the R object that gets created. For example, sending the JMP variable dt to R creates an R object named dt. The colon and double colon scoping operators (: and ::) are not valid in R object names, so these are converted as follows:

• A single colon scoping operator is replaced with a period (.).

  For example, sending nsref:dt to R creates a corresponding R object named nsref.dt.

- A double colon scoping operator (designating a global variable) is ignored.

  For example, sending `::dt` to R creates a corresponding R object named `dt`.

Using R Name() with R Send()

The R `Name()` option to R `Send()` has an argument that is a quoted string that contains a valid R object name. The JMP object sent to R becomes an R object with the name specified. For example:

```
R Send( jmp_var_name, R Name( "r_var_name") );
R Submit( "print(r_var_name)" )
```

Example

This example creates a variable x in the Here namespace, a variable y in the global namespace, and a variable z that is not explicitly referenced to any namespace. The variable z defaults to Global unless Names Default To Here(1) is on. These variables are then passed to R.

```
Here:x = 1;
::y = 2;
z = 3;

R Init(); // initiate the R connection

/* send the Here variable to R
Here:x creates the R object Here.x */
R Send( Here:x );

/* note that the JMP log labels the output with the original JMP variable
   reference Here:x */
R Submit( "print(Here.x)" );

R Send( ::y ); // ::y create the R object y
R Submit( "print(y)" );

// to use a different name for the R object, use the R Name() option
R Send( Here:x, R Name( "localx" ) );

/* The R Name option to the R Send() command creates the R object named
   "localx"" corresponding to the JMP variable "Here:x". Again the log shows
   the original corresponding JMP variable name. */
R Submit( "print(localx)" );
R Send( z ); // z creates the R object z
R Submit( "print(z)" );
```

Troubleshooting

Recording Output

On Windows, if you want to record output to the graphics window, send the following R code using R Submit( ).

```
windows.options( record = TRUE );
```

Character Vectors

A JMP list of strings is not the same as an R character vector. If you send a list of strings to R, it becomes a list of strings in R, not a character vector. You can use the R function Unlist to convert it:

```
R Init();
X = {"Character", "JMP", "List"};
R Send( X );

/* R output is:
[1] "list"
*/
R Submit( "class(X)" );

/* Object Y is now a character vector. The R output is:
[1] "character"
*/
R Submit( "Y<-unlist(X)
    class(Y)" );
R Term();
```

Element Names

A feature of an R list (called attributes) lets you associate a name with each element of the list. You can use the name to access that element instead of having to know the position of it in the list

In the following example, the list that is created in R has two elements named x and y that are created using the List() function of R. When your bring the R list into JMP and then send it back to R, the names are lost. Therefore in R, you cannot access the first matrix using pts$x. Instead, you must use the index using pts[[1]].

```
R Init();
R Submit("
  pts <- list(x=cars[,1], y=cars[,2])
  summary(pts)
");
```

```
JMP_pts = R Get( pts );

R Send( JMP_pts );
R Submit("
  Summary( JMP_pts )
");
R Term();
```

Examples

Sending a Data Table to R

This example initiates an R connection, sends a data table to R, prints it to the log, and closes the R connection.

```
R Init();
dt = Open( "$SAMPLE_DATA/Big Class.jmp", invisible );
R Send( dt ); // send the opened data table represented by dt to R;
R Submit( "print( dt )" );
R Term();
```

Creating Objects in R

This example initiates an R connection, creates an R object, retrieves the object into JMP, and closes the R connection.

```
R Init();
R Submit(
  "
  L3 <- LETTERS[1:3]
  d <- data.frame(cbind(x=1, y=1:15), Group=sample(L3, 15, repl=TRUE))
  "
);
R Get( d ) << NewDataView;
R Term();
```

Using R Functions and Graphics

This example initiates an R connection and plots the normal density function in R using the R graphics device,. Then the graph is retrieved from R and displayed in JMP. Finally, the R connection is closed.

```
R Init();
R Submit( "\[plot(function(x) dnorm(x), -5, 5, main = "Normal(0,1) Density")
  ]\" );
```

```
picture = R Get Graphics( "png" );
New Window( "Picture", picture );
Wait( 10 );
R Term();
```

Simple Matrix Addition in R

This example initiates an R connection, sends a matrix to R, creates a matrix in R, adds them together, returns the new matrix to JMP, and closes the R connection.

```
R Init();
X = J( 2, 2, 1 );
R Send( X );
R Submit(
  "
  X                                       #prints X to the log
  Y <- matrix(1:4, nrow=2, byrow=TRUE)   #makes a 2x2 matrix object Y
  Y                                       #prints Y to the log
  Z <- X + Y #matrix object Z is addition of X and Y
  "
);
Z = R Get( Z );
R Term();
Show( Z );
```

A Bootstrap Sample

See the file JMPtoR_bootstrap.jsl in the sample scripts folder for an example script.

This script performs a bootstrap simulation by using the JMP to R Project integration.

The script produces a JMP window that asked the user to specify the variable to perform bootstrapping over. Then the user selects a statistic to compute for each bootstrap sample. Finally, the data is sent to R using the R interface in JSL.

The boot package in R is used to call the boot() function and the boot.ci() function to calculate the sample statistic for each bootstrap sample and the bootstrap confidence interval.

The results are brought back to JMP and displayed using the JMP Distribution platform.

Work with Python

The JMP-Python integration allows JMP users to interact with a large amount of scientific, statistical, and data analysis tools written in Python. JMP implements this capability by loading the Python shared library into the running JMP process. JSL gives Python access to JMP.

The JMP installer does not contain Python. Several Python implementations are available. Visit https://www.python.org/download/alternatives to view the options.

JMP has several methods for finding an instance of Python installed on the computer. You should not need to set any environment variables or additional arguments to the JSL `Python Init()` statement. See "Troubleshooting the JMP Python Integration" on page 806 for more information.

The Python integration enables you to perform the following tasks:

- submit statements to Python from within a JSL script
- exchange data between JMP and Python
- display graphics produced by Python

Text output and error messages from Python appear in the log window.

Install Python

JMP 16 is a 64-bit program; only 64-bit Python is supported

- Python.org 3.6 through 3.9.1 distributions are supported on Windows and macOS. On Apple silicon, you will not have full functionality until third-party packages have been updated.
- Anaconda 3.6 distribution is supported on Windows only.
- Anaconda 3.7 distribution is not supported on Windows.
- The way Python is built from versions 3.6 to 3.7 changed. In Python 3.7, Python must be installed for all users. Otherwise, the libraries that the Python DLL depend on cannot be located at DLL load time. Other Python distributions might work fine installed for a single user or might also require a system-wide install.
- JMP does not currently support Python virtual environments.

Notes:

- Python is not distributed with JMP. Download Python from the Python web site at https://python.org.

- Several Python implementations are available. Visit
  https://www.python.org/download/alternatives to view the alternatives.

- On macOS, installing Python 3.6.x from https://python.org installs in
  /Library/Frameworks/Python.framework.

Required Python Packages

For the JMP-to-Python interfaces to work properly, you must install certain Python packages
and their prerequisites. In many Python installations, such as Anaconda, these packages are
already installed. If you install Python from Python.org, also install the following packages:

numpy   https://www.numpy.org

pandas   https://pandas.pydata.org

matplotlib   https://matplotlib.org

scipy   https://www.scipy.org

sqlite3   https://www.sqlite.org

PyQt5   https://pypi.org/project/PyQt5/

Use the standard Python update tools to install these packages for your Python configuration.
You must use the correct pip-tool for your installation. The pip-tool name is dependent on the
what product has installed Python. The installation instructions for Python have instructions
on using pip tools.

Troubleshooting the JMP Python Integration

Sometimes JMP has trouble locating the correct version of Python. The first thing to try is
running Python from the command line. If Python doesn't run from a terminal or console
window, something is wrong with the Python installation.

Note: Before you proceed with troubleshooting, verify that you're running a supported
version of Python. See "Install Python" on page 805.

How JMP Finds Python on Windows

JMP delays loading Python until a JSL-based script requires access to it. When JMP loads
Python, it needs to be able to find the Python shared library and any of the shared libraries
that the Python DLL depends on. Once the Python interpreter is loaded, Python needs to be
able to find the distribution's standard Python modules. To guide this process, JMP uses
registry entries, environment variables, and user-specified paths.

Python also looks for certain environment variables. Avoid using any user-specified paths or environment variables, and let JMP find Python in the registry. The following methods help JMP locate Python if the automatic location doesn't work:

- `JMP_PYTHON_MODULE_PATH` is a JMP-specific version of `PYTHONPATH`. This variable is the replacement for `Python Sys Path`. In `JMP_PYTHON_MODULE_PATH`, specify standard OS paths. Separate the paths with a colon on macOS. Separate the paths with a semicolon on Windows.

- `JMP_LIB_PYTHON_PATH` is a JMP-specific version of `PYTHONHOME`. The variable contains the full path to the Python shared library, or the parent directory to the shared library.

Windows finds Python in the following order of precedence:

1. JMP looks at the environment variables `Python Sys Path` (deprecated), `PYTHONINSTALLPATH` (deprecated), and `JMP_PYTHON_MODULE_PATH`, in that order. JMP uses the first variable that is found.

2. JMP checks `JMP_LIB_PYTHON_PATH` for a path to the Python shared library.

3. JMP checks for options that are specified in the `Python Init()` or `Python Connect()` function calls. `Path()` corresponds to the location of the Python shared library. `Use Python Version()` allows you to specify the version of Python to look for.

   `Python Sys Path` provides the mechanism to set Python's `sys.path` variable. If present, script parameters take precedence over the environment variables. There is one caveat. After JMP loads the Python shared library, the library is unloaded only when JMP shuts down. As a result, the `Path()` portion is only effective on the first load of Python, and does not allow for running different versions of Python in different scripts without shutting down JMP and restarting.

4. JMP reads the Windows registry to determine if Python has been installed. It looks at appropriate locations based on whether the installation was for an individual user or for all users on the Windows machine. If you have specified a Python version, other versions of Python will not be considered.

If JMP cannot locate Python, try the following option in the `Python Init()` expression:

```
Python Init( Init Trace( "TRUE" ) );
```

JMP prints details to the log about where it detects Python, including paths and registry entries for Python.

Typically, JMP determines the equivalent to the `PYTHONHOME` environment variable automatically if it is defined in the Windows registry.

Computer\\<root>\\SOFTWARE\\Python\\PythonCore\\<version>\\InstallPath

<root> is either the root key HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER depending on where and how you installed Python. *<version>* is the Python version number.

If multiple versions of Python are installed on your computer, you must specify which installation for JMP to use.

- The easiest method is to specify the version using the `Use Python Version()` argument in `Python Init()` or `Python Connect()` functions.
- Use the `Path("path to an installed Python DLL or shared library")` argument with the `Python Init()` or `Python Connect()` function.
- Define the `JMP_LIB_PYTHON_PATH` environment variable as described below.

Change the Default Python Install Location on Windows

Create the variable in your system environment variables using the Control Panel.

1. Select **Start > Control Panel > System > Advanced system settings**.
2. Click **Environment Variables**.
3. In the System variables pane, click **New**.
4. Type `JMP_LIB_PYTHON_PATH` for the Variable name.
5. Type the path to the Python<version>.dll file, for example:

   `C:/Program Files/Python/<version>/Python<version>.dll`
6. Click **OK** and click **OK** again to close the System Properties window.

macOS: Override How JMP Finds Python

Locating Python on the Mac works the same as on Windows except for the step of looking in the registry. JMP looks in a couple of typical installation directories so see if it can locate Python. Python typically installs in /Library/Frameworks/Python.framework. A compatible version of Python located in the standard system location should work without any changes to environment variables or parameters passed to JSL.

If Python is installed in an alternate location, set configuration variables in your shell environment as needed. JMP on macOS uses the same environment variables as described in "How JMP Finds Python on Windows" on page 806.

JMP_LIB_PYTHON_PATH is the path to the Python shared library libpython3.x.dylib, where *x* is the minor version. If environment variables are needed, you typically only need to set JMP_LIB_PYTHON_PATH.

If JMP is unable to find install Python packages, you might also need to set the variable JMP_PYTHON_MODULE_PATH. The JMP_PYTHON_MODULE_PATH contains the same paths that the Python call print(sys.path) outputs. When you set an environment variable, use host OS path conventions. When the variable is passed in JSL, use a JSL list.

One way is to have Python tell you what it thinks its sys.path should be. In the example in this section, the same Python commands can be run on Windows to determine sys.path.

In the terminal or console window, enter the following boldface commands to get a string that specifies the location of the Python packages:

```
MDz320:~ shlori$ python --version
Python 3.6.4 :: Anaconda, Inc.
MDz320~ shlori$ python3
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 12:04:33) [GCC 4.2.1
    Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin Type "help",
    "copyright", "credits" or "license" for more information.
>>> import sys
>>> print(sys.path)
['', '/anaconda3/lib/python36.zip', '/anaconda3/lib/python3.6',
    '/anaconda3/lib/python3.6/lib-dynload',
    '/anaconda3/lib/python3.6/site-packages',
    '/anaconda3/lib/python3.6/site-packages/aeosa']
```

Test Your Setup

To test that your computer is able to run JSL scripts that use Python, run the following script:

```
Python Init();
Python Submit( "\[basket = ['apple', 'orange', 'pear']
print(basket)]\" );
Python Term();
```

You should see the following output in the log:

```
['apple', 'orange', 'pear']
0
```

JMP-to-Python Interfaces

JMP provides the interfaces to access Python that are described in the following sections. The basic execution model is to first initialize the Python connection, perform the required Python operations, and then terminate the Python connection. In most cases, these functions return 0 if the Python operation was successful or an error code if it was not. If the Python operation is not successful, a message is written to the log. The single exception to this is Python Get(), which returns a value.

Python JSL-Scriptable Object Interfaces

The Python interfaces are also scriptable using a Python connection object. A scriptable Python connection object can be obtained using the Python Connect() function.

```
// return a Python connection scriptable object
PythonConnection = Python Connect();

// return 1 if a Python connection is active. Otherwise, returns 0.
x = PythonConnection << Is Connected;
Show( x );
PythonConnection << Disconnect;
  x = 1;
  0
```

Equivalent Data Types for Python Send()

Table 14.5 shows what JMP data types can be exchanged with Python using the Python Send() function. Sending lists or associative arrays to Python recursively examines each element of the list or associative array and sends each base JMP data type. Nested lists and associative arrays are supported.

Table 14.5 Equivalent JMP and Python Data Types for Python Send()

JMP Data Type	Python Data Type
Boolean	Boolean
Data Table	Pandas.DataFrame
Associative Array	Dictionary

Table 14.5 Equivalent JMP and Python Data Types for Python Send() *(Continued)*

JMP Data Type	Python Data Type
Numeric	Float
Matrix	Float Matrix
List	List
String	Unicode String

Python Send() Example

```
Python Init();
X = 1;
Python Send( X );
S = "Report Title";
Python Send( S );
M = [1 2 3, 4 5 6, 7 8 9];
Python Send( M );
Python Submit( "\[
print(X)
print(S)
print(M)
]\" );
Python Term();
   1.0
   Report Title
   [[ 1.   2.   3.]
    [ 4.   5.   6.]
    [ 7.   8.   9.]]
   0
```

Equivalent Data Types for Python Get()

Table 14.6 shows which JMP data types can be exchanged with Python using the Python Get() function. Getting lists from Python recursively examines each element of the list and gets each base Python data type. Nested lists are supported.

Table 14.6 Equivalent Python and JMP Data Types for Python Get()

Python Data Type	JMP Data Type
Array	Numeric Matrix
Date and time	Numeric

Table 14.6 Equivalent Python and JMP Data Types for Python Get() *(Continued)*

Python Data Type	JMP Data Type
Pandas.DataFrame	Data Table
Dictionary	Associative Array
Graph	Picture Object
Integer	Integer
List	List of converted Python data types
Matrix	Numeric Matrix
Set	List
Unicode String	String

Python Get() Example

```
Python Init();
x1 = [1, 2, 3];
Python Send( x1 );
x2 = Python Get( x1 );
Show( x1, x2 );
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
Python Send( dt1 );
dt2 = Python Get( dt1 );
dt2 << New Data View;
Close( dt1 );
Python Term();
```

JMP Scoping Operators and Python

A JMP object sent to Python with Python Send() uses the same JMP reference as the name of
the Python object that gets created. For example, sending the JMP variable dt to Python
creates a Python object named dt.

The colon and double colon JSL scoping operators (: and ::) are not valid in Python object
names, so the colons are converted as follows:

- A single colon scoping operator is replaced with an underscore (_).

  For example, sending nsref:dt to Python creates a corresponding Python object named
  nsref_dt.

- A double colon scoping operator (designating a global variable) is ignored.

For example, sending `::dt` to Python creates a corresponding Python object named `dt`.

Note: The `Python Name` option for `Python Send()` has a quoted string argument that contains a valid Python object name. The JMP object sent to Python becomes a Python object with the specified name.

Python Name() and Python Send() Examples

The following example creates the `jmp var name` variable, assigns it to the Python name `python_var_name`, submits the `print` statement, and closes the connection.

```
Python Init();
jmp var name = 25;
Python Send( jmp var name, Python Name( "python_var_name" ) );
Python Submit( "print(python_var_name)" );
Python Term();
```
25.0

0

The following example creates a variable *x* in the Here namespace, a variable *y* in the global namespace, and a variable *z* that is not explicitly referenced to any namespace. The variable *z* defaults to Global unless `Names Default To Here(1)` is on. These variables are then passed to Python.

```
Here:x = 1;
::y = 2;
z = 3;

Python Init(); // initiate the Python connection

/* send the Here variable to Python
Here:x creates the Python object Here_x */
Python Send( Here:x );

Python Submit( "print(Here_x)" );

Python Send( ::y ); // ::y creates the Python object y
Python Submit( "print(y)" );

// to use a different name for the Python object, use the Python Name() option
Python Send( Here:x, Python Name( "localx" ) );

/* The Python Name option to the Python Send() command creates the Python
   object named "localx", which corresponds to the JMP variable "Here:x". */
Python Submit( "print(localx)" );
```

```
Python Send( z ); // z creates the Python object z
Python Submit( "print(z)" );
Python Term();
    1.0

    2.0

    1.0

    3.0

    0.0
```

Additional Python Integration Examples

Send a Data Table to Python

This example initiates a Python connection, sends a data table to Python, prints it to the log, and closes the Python connection.

```
Python Init();
dt = Open( "$SAMPLE_DATA/Big Class.jmp", invisible );
Python Send( dt ); // send the opened data table represented by dt to Python
Python Submit( "print( dt )" );
Python Term();
            name     age   sex   height   weight
    0       KATIE    12    F        59       95
    1      LOUISE    12    F        61      123
    2        JANE    12    F        55       74
    ...
    0
```

Create Objects in Python

This example initiates a Python connection, creates a Python object, retrieves the object into JMP, and closes the Python connection.

```
Python Init();
Python Submit("\[
import numpy as np
import pandas as pd
# Generate basic series
ss = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
print(ss)
sn = pd.Series(np.random.randn(5))
print(sn)

# Generate Data Frames from Series
```

```
s1 = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
print(s1)
dfs1 = pd.DataFrame(s1)
print(dfs1)

d1 = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
      'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
print(d1)
dfd1 = pd.DataFrame(d1)
print(dfd1)

d2 = {'one' : [1., 2., 3., 4.],
      'two' : [4., 3., 2., 1.]}
print(d2)
dfd2 = pd.DataFrame(d2)
print(dfd2)
]\"
);
ss = Python Get( ss );
Show( ss );
sn = Python Get( sn );
Show( sn );
dfs1 = Python Get( dfs1 );
dfs1 << New Data View;
dfd1 = Python Get( dfd1 );
dfd1 << New Data View;
dfd2 = Python Get( dfd2 );
dfd2 << New Data View;
Python Term();
    a    -0.384649
    b     0.455864
    c    -2.214405
    d     0.260955
    e     1.342286
    dtype: float64
    ...
```

Matrix Operations in Python

In JMP, a simple matrix looks like this:

```
xx = [1 2 3, 4 5 6, 7 8 9 ];
```

In numpy, the matrix is an array:

```
xx = numpy.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

In order to submit a matrix as a numpy array, you must translate the JMP matrix.

```
q = "new = numpy.array ([";
```

```
For( i = 1, i <= N Items( xx ), i++,
    zz = Matrix( xx[i] );
    q = q || Char( zz ) || ",";
);
q = q || "])";
Show( q );

Python Submit( "import numpy
" || q );
Python Submit( "print(new)" );
```

Work with Microsoft Excel

You can script the Profiler interface to Microsoft Excel, although not the Transfer to JMP interface. Here is the basic syntax:

```
excel_obj = Excel Profiler(
    Workbook( "excel_workbook_path" ),
    Model( "name_of_model" )
);
```

The `Model` argument is optional. If you supply only the workbook, you are prompted to select the model. You do not need to specify the worksheet, because the model is found no matter where it is located.

Once you have an Microsoft Excel Profiler object, you can send the following messages to run the JMP prediction profiler using the Microsoft Excel model. For example:

```
excel_obj <<Prediction Profiler( 1 );
```

Parse XML

JSL has several commands available to parse XML.

```
Parse XML( string, On Element( "tagname", Start Tag( expr ), End Tag( expr
    ) ) );
```

parses an XML expression using the `On Element()` expression for specified XML tags.

```
value = XML Attr( "attribute name" );
```

extracts the string value of an XML argument when evaluating a `Parse XML()` expression.

```
value = XML Text();
```

extracts the string text of the body of an XML tag when evaluating a `Parse XML()` expression.

Example of Parsing XML

Suppose that a Microsoft Excel file contains one row of data from Big Class.jmp. The file is saved as the valid XML document BigclassExcel.xml, shown here and also saved in the JMP Samples/Import Data folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
 xmlns:o="urn:schemas-microsoft-com:office:office"
 xmlns:x="urn:schemas-microsoft-com:office:excel"
 xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
 xmlns:html="http://www.w3.org/TR/REC-html40">
 <Worksheet ss:Name="Bigclass">
  <Table ss:ExpandedColumnCount="5" ss:ExpandedRowCount="41"
   x:FullColumns="1"
   x:FullRows="1">
   <Row>
    <Cell><Data ss:Type="String">name</Data></Cell>
    <Cell><Data ss:Type="String">age</Data></Cell>
    <Cell><Data ss:Type="String">sex</Data></Cell>
    <Cell><Data ss:Type="String">height</Data></Cell>
    <Cell><Data ss:Type="String">weight</Data></Cell>
   </Row>
   <Row>
    <Cell><Data ss:Type="String">KATIE</Data></Cell>
    <Cell><Data ss:Type="Number">12</Data></Cell>
    <Cell><Data ss:Type="String">F</Data></Cell>
    <Cell><Data ss:Type="Number">59</Data></Cell>
    <Cell><Data ss:Type="Number">95</Data></Cell>
   </Row>
  </Table>
 </Worksheet>
</Workbook>
```

The following script reads BigclassExcel.xml and creates a JMP data table with the information in it. This script, named ParseXML.jsl, is in the JMP Samples/Scripts folder.

```
file contents = Load Text File( "$SAMPLE_IMPORT_DATA/BigclassExcel.xml" );
Parse XML( file contents,
  OnElement(
    "urn:schemas-microsoft-com:office:spreadsheet^Worksheet",
    StartTag(
      sheetname = XML Attr(
        "urn:schemas-microsoft-com:office:spreadsheet^Name",
        "Untitled"
      );
      dt = New Table( sheetname );
      row = 1;
```

```
            col = 1;
        )
    ),
    OnElement(
        "urn:schemas-microsoft-com:office:spreadsheet^Row",
        StartTag(
            If( row > 1, // first row has column names
                dt << Add Rows( 1 )
            )
        ),
        EndTag(
            row++;
            col = 1;
        )
    ),
    OnElement( "urn:schemas-microsoft-com:office:spreadsheet^Cell", EndTag(
    col++ ) ),
    OnElement(
        "urn:schemas-microsoft-com:office:spreadsheet^Data",
        EndTag(
            data = XML Text( collapse );
            If( row == 1,
                New Column( data, Character( 10 ) ), // first row has column names
                Column( col )[row - 1] = data // and other rows have data
            );
        )
    )
)

);
```

OLE Automation

Most of JMP can be driven through OLE automation. Please see the Automation Reference.pdf
at https://www.jmp.com/en_us/support/jmp-documentation.html for more information about
automating JMP. This document introduces how to automate JMP through Visual Basic and
using Visual C++ with MFC. It also contains more information about the methods and
properties that JMP exposes to automation clients like Visual Basic and Visual C++.

The JMP Samples/Automation folder contains several sample Visual Basic .Net, and Visual C#
.Net files.

Share Reports on JMP Live

Write JSL scripts to automate the generation of JMP analyses and publish them to JMP Live.

JMP Live Prerequisites

Before you can use JSL to publish reports to JMP Live, do the following:

1. Get an API key (if required by a JMP Live administrator). See Get an API Key.

2. Establish a connection to a JMP Live server. See *Using JMP*. When you add a JMP Live server, you need to enter your API key (if required).

   When you establish a connection, a JMP Live Connection is created in JSL:

   ```
   liveconnection = New JMP Live( Connection( "My JMP Live" ), Prompt( If Needed
     ));
   ```

   where "My JMP Live" is the name of the JMP Live server connection you created. If authentication is required, an interactive prompt appears. You can change the Prompt to No, but if authentication is required, the connection fails.

Get an API Key

Before you can script reports and publish them to JMP Live, you might need an API key. JMP Live administrators determine whether an API key is required for scripting.

An API key is unique for a user and a JMP Live server. That means that if you obtain a new API key, it overrides the previous one, and any scripts using the previous API key will need to be updated to use the new API key.

To obtain an API key:

1. Log into JMP Live.

2. At the top, click *your user icon* and select **Settings**.

3. Click **Generate API Key**. Make a note of your key.

Publish a Single Report to JMP Live

Publish a single report to JMP Live using a JMP Live Report.

The following example creates a Bivariate report and an empty JMP Live Report called webreport, adds the Bivariate report to the webreport, and publishes the webreport to JMP Live:

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
// Create a Bivariate report.
biv = Bivariate( X( :Height ), Y( :Weight ));
// Create an empty JMP Live Report.
webreport = New Web Report();
/* Add the Bivariate report to the webreport. A title is required and a
   description is optional.*/
webreport << Add Report( biv, Title( "Bivariate Report" ), Description( "A
   test of the new scripting" ));
/* Publish the webreport to JMP Live as a private report. Use the JMP Live
   Connection you established and return a JMP Live object called liveresult.
   */
liveresult = liveconnection << Publish( webreport );
```

You can use JSL to make the report visible to everyone or share the report with groups in JMP Live. See the Set Public and Share With Groups messages in the JMP Scripting Index at Help > Scripting Index.

Work with a JMP Live Result

The Publish function returns a JMP Live Result (liveresult). Use JSL scripting functions on a JMP Live Result to get information, such as whether the publishing succeeded or failed, failure codes, and more:

```
// Return the outcome of the last action.
liveresult << Succeeded ();
// Return any error messages from the last action.
liveresult << Get Error Message();
```

The JMP Live Result can be made scriptable and turned into a JMP Live Report (for a single report) or a JMP Live Folder object (for multiple reports) so that you can send commands to it. The following example changes a JMP Live Result into a scriptable JMP Live Report and then changes the report access from private to public:

```
livereport = liveresult << As Scriptable();
livereport << Set Public( 1 );
```

See the JMP Live Report object in the JMP Scripting Index at Help > Scripting Index.

Publish Several Reports to a JMP Live Folder

Publish several reports to JMP Live by creating a JMP Live Folder and adding the JMP Live Report to the folder. A JMP Live Report object can hold a single report or multiple reports.

The following example creates a scriptable JMP Live Folder and a JMP Live Report called webreport, adds a Bivariate and a Oneway report to the webreport, and adds the reports to a folder:

```
/* Create a folder in JMP Live. A title is required and a description is
   optional. Use the JMP Live Connection you established and return a JMP Live
   object called liveresult. */
liveresult = liveconnection << Create Folder( Title( "Test Folder"
  ),Description( "Folder created for script testing" ));
// Make the liveresult scriptable.
folder = liveresult << As Scriptable();
// Create an empty JMP Live Report.
webreport = New Web Report();
// Add a Bivariate report to the webreport.
webreport << Add Report( biv, Title( "Bivariate Report" ));
// Add a Oneway report to the webreport.
webreport << Add Report( oneway, Title( "Oneway Report" ));
// Add the reports to the folder.
liveresult = folder << Add Reports To Folder( webreport );
```

You can use JSL to share the folder with JMP Live groups. See the Share With Groups object in the JMP Scripting Index at **Help** > **Scripting Index**.

Delete Reports in a JMP Live Folder

You can delete reports and folders using JSL. Folders must be empty before they can be deleted.

```
returncode = liveconnection << Delete Report( report << Get ID() );
returncode = liveconnection << Delete Folder( folder << Get ID() );
```

In both of these cases, a value of 1 (True) or 0 (False) will indicate if the operation was successful.

Replace the Contents of a JMP Live Report

You can replace the contents of an existing report rather than creating a new report with updated content. You need to know the ID of the report you want to replace. If you have the report object available, you can use the Get ID( ) function to return the ID.

The following example creates a new JMP Live Report called webreport2, adds an updated Oneway report, and replaces an existing Oneway report using the report ID:

```
// Create an empty JMP Live Report.
webreport2 = New Web Report();
/* Add a Oneway report to webreport3. A title is required and a description is
   optional. */
webreport2 << Add Report( oneway, Title( "New Oneway Report" ), Description(
   "Replace a report" ));
/* Use the JMP Live Connection you established to replace the report with the
   specified ID with webreport2. */
liveconnection << Replace( webreport2, ID(report << Get ID() ));
```

Search for a JMP Live Report or Folder ID

For many JSL functions, you need to know the report or folder ID. Using JSL, you can search by keyword (in the title or description) or by the publisher of the report or folder. If your search may return many results, you can specify how many results to return at a time.

The following example searches all reports that contain the word "Production" in the title or description, and are published by a user with the display name "dianesmith". It also specifies that the search will return up to ten items at a time.

```
/* Search for reports with the keyword Production that are published by the
   user dianesmith. Return no more than 10 results at a time. */
liveresult = liveconnection << Find Reports( Search( "Production" ),
   Publisher( "dianesmith" ), ( PageSize( 10 ));
// Change the liveresult into JMP Live Reports and make them scriptable.
livereports = liveresult << As Scriptable();
// Show how many results are returned.
show( livereports << Get Number Of Items());
// Return the first report in list of results.
report1 = livereports[ 1 ];
// Show the title of the first report.
show( report1 << Get Title());
```

In the example above, if you might have more than ten results, you can page through them using the Next and Previous functions:

```
// Put the next 10 results into the next page.
nextpage = livereports << Next();
// Show the current page number.
show( livereports << Get Current Page Number());
// Get the first item in the list of items in the next page.
next1 = nextpage[ 1 ];
// Show the title of the first report.
show( next1 << Get Title());
```

```
// Show the number of items in the current page.
show( nextpage << Get Number Of Items());
```

If `Next()` finds no further pages, an error appears, which you should catch to proceed with further processing.

Tip: If you know the number of pages to expect, you can jump directly to a page. See the `Get Page` function in the JMP Scripting Index at **Help** > **Scripting Index**.

Update Data in JMP Live Reports

If the data behind a JMP Live report changes, you can update the data and regenerate the report. You can update the data for a standalone report (using the report ID) or for all the reports in a folder (using the folder ID).

The following example updates data for an individual report. Suppose that in the Big Class.jmp data table, a subject's age changed and you want to update an associated age report on JMP Live.

```
dt:Age[1] = 77;
/* Use the JMP Live Connection you established to update the data for the
   report with the specified ID based on the updated Big Class.jmp data table.
   */
updateresult = liveconnection << Update Data( report << Get ID( ), Data( dt,
   "Big Class.jmp" ));
// Show the outcome of the update.
show( updateresult << Succeeded );
```

Similarly, you can also specify the folder that the report resides in:

```
updateresult = liveconnection << Update Data( folder << Get ID( ), Data( dt,
   "Big Class.jmp" ));
```

Only reports in the folder using the specified data table update.

Work with Other Content in JMP Live Reports

Primarily in JSL, you work with JMP Live folders and reports. However, you can publish images to JMP Live reports. In JSL, you can use a generic object called a JMP Live Post to manage images or other content that does not fall under reports or folders. See the `JMP Live Post` function in the JMP Scripting Index at **Help** > **Scripting Index**.

Communicate with REST Web Services

Representational state transfer (REST) or RESTful web services is a way of providing interoperability between computer systems on the Internet. Use JSL to communicate with a RESTful application program interface (API) that uses HTTP requests to GET, POST, PUT, and DELETE data. For example, you can get data from a web service and return it as a data table. Some web services are Amazon Web Services (AWS), DropBox, and Text-Processing.com.

Notes:

- Strings that contain "&" are truncated. "+" is converted to a space.

- New HTTP Request() supports redirects, for example, from HTTP to HTTPS.

Examples of HTTP Requests

The following examples communicate with the National Aeronautics and Space Administration (NASA) API. You must request a key from https://api.nasa.gov/index.html#apply-for-an-api-key and assign it to the key variable to run the scripts successfully.

Write Data to the Log

```
url = "http://technology.nasa.gov/api/query/patent/all";
request = New HTTP Request( URL( url ), Method( "Get" ) );
data = request << Send();
Write( data );
    {"results":[["59fa04859600022b4d2e076f","LAR-TOPS-48","All-Organic
        Electroactive Device","NASA's Langley Research Center offers you an
        all-organic electroactive device system fabricated with single-wall
        carbon nanotube (SWCNT)....}
    // JSON output written to the log
```

Notes:

- The query string is at the end of the URL, following "?". For example, in https://www.google.com/search?q=jmp, "q=jmp" is the query string. New HTTP Request() automatically URI-encodes key/value pairs in a query string (and FORM).

- When you put an associative array in either a query string or a form, the key/value pairs get URI encoded. Spaces, ampersands, and other characters get encoded to ASCII, and the ampersand is escaped. The encoding is a "%" encoding.

- The web service defines the valid key/value pairs that it accepts. Refer to the API documentation for valid key/value pairs.

Return Data as an Associative Array and Write it to the Log

```
url = "http://technology.nasa.gov/api/query/patent/all";
request = New HTTP Request( URL( url ), Method( "Get" ) );
json = request << Send();
jsl_json = Parse JSON( json );
// turns JSON data into a JSL associative array
results = jsl_json["results"];
Write( results );
    {{"59fa04859600022b4d2e076f", "LAR-TOPS-48", "All-Organic Electroactive
        Device", "NASA's Langley Research Center offers you an all-organic
        electroactive device system fabricated with single-wall carbon nanotube
        (SWCNT)....}}
    // associative array
```

Return Data as a JMP Data Table

```
url = "http://technology.nasa.gov/api/query/patent/all";
request = New HTTP Request( URL( url ), Method( "Get" ) );
json = request << Send();
jsl_json = Parse JSON( json );
json = As JSON Expr( jsl_json["results"] );
dt = JSON To Data Table(
    json,
    JSON Settings(
        Stack( 0 ),
        Col(
            "/root",
            Column Name( "root" ),
            Fill( "Use Once" ),
            Type( "Pandas Values" ),
            Format( {"Best"} ),
            Modeling Type( "Continuous" )
        )
    )
);
dt << Set Name( "NASA Patents" );
```

Figure 14.5 Data Imported in a Data Table

OAuth 2.0 for Web APIs

OAuth 2.0 is an authorization standard for accessing personal user information through web APIs. An authorization standard is an agreed upon set of rules by many major web companies that are responsible for authorizing who can access information, and what scope of information they can access. This authorization is granted in the form of a limited time access *token* (a unique identifier specific to the individual and the scope of information they can access). Due to this two-pronged approach of limiting both users and scopes, OAuth is the current standard for securing personal information. Web sites such as Google, Salesforce, Reddit, and Dropbox use OAuth 2.0.

Since OAuth secures your personal information, it can appear to be fairly complex. OAuth is even more complicated due to the variety of ways it is implemented, which are called *grant types*. We'll walk through the different ways you can get an OAuth token in JSL. Please note that not all methods will work with all companies, so you'll have to do a little research to find out which grant type to use. Usually, the company will have documentation of this either with their APIs or possibly in a separate OAuth section.

We recommend that you familiarize yourself with the OAuth protocol at the official website (https://oauth.net/2/).

Note: Only OAuth 2.0 is supported in JSL. OAuth 1.0 and 1.0a are still used by a handful of companies (such as Twitter) but are largely regarded as inferior to 2.0.

Security

OAuth tokens can contain sensitive information. Depending on the grant type, you might need to provide your password or a client secret. JMP automatically handles this information for you.

Information that Google uses to authenticate your account is stored in C:/Users/<user name>/AppData/Roaming/SAS/JMP/OAuthTokens.jmp. (Your Google password is not stored.) After using an OAuth token on your personal account, you can delete all columns except for User and Client ID. Keep in mind that if you need to share your script, others won't have the User and Client ID columns saved on their machine. We recommend moving this information to an encrypted JSL script before sharing the OAuth script.

Using Postman for Redirect URLs

JMP does not provide a redirect URL for free. However, using Postman, it is easy to set one up yourself. Some examples that follow use Postman to redirect requests.

1. Go to https://www.getpostman.com.
2. Click the orange sign-in button in the top right corner.
3. Sign in with your Google account, or proceed to create your own account.
4. You can then use https://app.getpostman.com/oauth2/callback as a redirect URL.

Authorization Code Grant

The Authorization Code grant usually sends *refresh tokens*. Refresh tokens are unique in that they act as a sort of pseudo, permanent access token. Although you won't access sensitive information with them, a refresh token is all you need to get more access tokens. Treat it like you would your password.

The following code snippet creates an OAuth token using the Authorization Code grant. Notice this requires `Request Auth()`, `Client Secret()`, and `Redirect URL()`. Remember, after you make the token once, you can remove everything but the User and Client ID from OAuthTokens.jmp.

```
token = New OAuth2 Token(
```

```
    User( "Test User" ),
    Client ID( "12ab" ),
    Client Secret( "3456dEfG" ),
    Request Auth(
        Scope( "history" ),
        Auth URL( "https://example.com/services/oauth2/authorize" ),
    ),
    Redirect URL( "https://www.getpostman.com/oauth2/callback" ) ,
    Token URL( "https://example.com/services/oauth2/token" )
);
```

Note: See your API documentation for more information about how to get values such as the client secret and token URL.

After you run this code once, clean it up and make it secure.

```
token = New OAuth2 Token(
    User( "Test User" ),
    Client ID( "12ab" ),
);
```

Existing Refresh Tokens

You might already be familiar with OAuth and have saved a refresh token to your script. The following snippet creates an OAuth token with that existing token.

```
token = New OAuth 2 Token (
    User( "yourgoogleaccount@gmail.com" ),
    Refresh Token( "1a2b3c4e5F" ),
    Token URL( "https://www.example.com/oauth2/token" ),
    Client ID( "12ab" ),
    Client Secret( "3456dEfG" )
);
```

After you run this code once, clean it up and make it secure.

```
token = New OAuth2 Token(
    User( "yourgoogleaccount@gmail.com" ),
    Client ID( "12ab" )
);
```

Implicit Grant

The Implicit grant type doesn't require any sensitive information. If you're going to share a script with other people, it would be easiest to use this grant type. Note, however, that because sensitive information isn't required, services are less likely to implement this flow.

The following code snippet creates an OAuth token using the Implicit grant. This requires `Redirect URL()`.

```
token = New OAuth2 Token(
   User( "Implicit Flow"),
   Client ID( "123abc" ),
   Redirect URL( "https://www.getpostman.com/oauth2/callback") ,
   Token URL( "https://example.com/services/oauth2/token" )
);
```

Resource Owner Grant

The Resource Owner grant is the only type that doesn't require a redirect URL. If setting up Postman (the API development platform) is proving difficult, try this grant type.

The following snippet will create an OAuth token using the Resource Owner grant. This requires `Client Secret()` and `Password()`.

```
token = New OAuth2 Token(
   User( "yourgoogleaccount@gmail.com" ),
   Password( "wordP@ss0192" ),
   Client ID( "1234abcd" ),
   Client Secret( "5678EfGh" ),
   Token URL( "https://example.com/services/v1/token" )
);
```

After you run this code once, clean it up and make it secure.

```
token = New OAuth2 Token(
   User( "yourgoogleaccount@gmail.com" ),
   Client ID( "1234abcd" ),
);
```

Using a Token

In order to access a web API that requires OAuth, pass the token to `HTTP Request()` using the OAuth2 field.

```
request = HTTPRequest(
   URL( apiEndpoint ),
   Method( "POST" ),
   OAuth2( token ),
);
```

OAuth 2.0 Resources

The official website for OAuth 2.0 is https://oauth.net/2/.

https://auth0.com/docs/protocols/oauth2 provides more information about the OAuth 2 protocol.

The OAuth 2.0 standard, RFC6749, is at https://tools.ietf.org/html/rfc6749.

Google provides comprehensive documentation and examples of OAuth 2.0 at https://developers.google.com/identity/protocols/OAuth2.

Communicate with SAS Cloud Analytic Services

SAS Viya uses Cloud Analytic Services (CAS) code to perform calculations in the cloud. This feature provides a way for SAS Viya users and JMP users to collaborate. You can write CAS code in JMP to import data from and export data to the CAS server, convert CAS JSON text to a JMP data table, get a list of available CAS libraries, and more.

- See the SAS CAS documentation at https://developer.sas.com/apis/cas/rest/current/apidoc.html for more information.
- The SAS CAS documentation for CAS actions cover CASL, R, Python, Lua.

This section covers the following topics:

- "About CAS Actions"
- "Connect to the CAS Server"
- "Create and Submit a CAS Action"
- "Create and Submit a CAS DATA Step Action"
- "Create a New CAS Server"
- "Import Data from the CAS Server"
- "View a CAS Dataset"
- "Get a List of Available CAS Libraries"
- "Export Data from a Data Table to a CAS Server"
- "Convert a CAS Table to a Data Table"
- "Delete a CAS Table"
- "Remove a CAS Table"
- "Verify the CAS Server Connection"
- "Get Details about the Current Connection"
- "Disconnect from the CAS Server"
- "Terminate All CAS Sessions"

About CAS Actions

The smallest unit of work for the CAS server is a CAS action. CAS actions can load data, transform data, compute statistics, perform analytics, and create output.

You configure each action by specifying a set of input parameters. Running a CAS action on the CAS server processes the action's parameters and the data, and creates an action result. See *CAS Actions* at https://developer.sas.com/apis/cas/actions.html.

The JSON parameters are specified in the CAS documentation *CAS REST API*. See https://developer.sas.com/apis/cas/rest/current/apidoc.html. For JSON, you can look at the Lua format and make the appropriate adjustments.

For example, in Figure 14.6, notice the difference between the JSON syntax and the Lua syntax. JSON uses the `"name" : value` format. Lua uses the `name = value` syntax.

Figure 14.6 Differences Between JSON and LUA

JSON Syntax   LUA Syntax

```
{                                              {
    "attrTable": "string",                         attrTable="string",
    "attrs": [ {                                   attrs={ {
        column": "string",                             column="string",
        key": "string",                                key="string",
        value": "string" |                             value="string" |
                number                                         64-bit integer |
}, {...} ],                                                    integer |
...                                                            double |
}                                                          binary-large-object
                                                      }, {...} },
                                               ,,,
                                               }
```

Connect to the CAS Server

You can show the login window and prompt the user to provide the URL for the CAS server and their login information.

```
CAS Connect( Prompt( "Never"|"Always"|"IfNeeded" ) );
```

To add the login details to the script, follow this example:

```
CAS Connect( URL( URL ), Username( username ), Password( password ) );
```

Information about the CAS server appears in the log after the user logs in.

To specify a trusted certificate for https connections to the CAS server, use the optional Certificates argument.

```
CAS Connect( URL( URL ), Certificates( string ) );
```

Create and Submit a CAS Action

```
New CAS Action( Action( "builtins.echo" ), JSON( echo ) );
```

Action is the JSL variable name. builtins.echo is the name of the CAS action that SAS created. echo contains the JSL arguments.

Create and Submit a CAS DATA Step Action

```
New CAS DATA Step action( Code( code ) );
```

code defines the contents of the DATA step. Action is the JSL variable name.

```
cas = Current CAS Connection();
code =
"\[
  data temp;
  x = 9.1; y = 6; z = sqrt(x**2 + y**2);
  A = "SAS"; B = "Statistics";
  put _ALL_;                 /* display all variables and values */
  run;
]\";
action = New CAS DATA Step action( Code( code ) );
cas << Submit( action );
```

Create a New CAS Server

New CAS Server() creates a new CAS server.

```
url = "URL";
cas = New CAS Server( Connect( URL( url ), Prompt(
  "Never"|"Always"|"IfNeeded" ) ) );
```

Import Data from the CAS Server

CAS Import Data() imports data from the CAS server:

```
CAS Import Data( libref.dataset, <Private|Invisible>, <"Use Labels for Var
  Names">);
```

- invisible hides the data table from view but shows it in the JMP Home Window. private hides the data table completely. Specify private if the user doesn't need to interact with the data table.

- "Use Labels for Var Names" indicates that the table name and the column names will be set to the SAS dataset label and the column labels. The appropriate column property, SAS Name SAS Label, is used.

View a CAS Dataset

CAS datasets are on the CAS file system. Datasets in memory are not listed.

To see the datasets on the CAS server that you want to import, use CAS Get Data Sets() after you connect to the server.

The following example shows how to list datasets that are only in casuser:

```
datasets = CAS Get Data Sets( casuser ); // quote the argument
Show( datasets );
```

The following example shows how to lists datasets that you have permissions to view:

```
datasets = CAS Get Data Sets();
Show( datasets );
```

Get a List of Available CAS Libraries

Libraries are namespaces, just like SAS libraries, librefs, or libnames. Use CAS Get Libraries() to get libraries from the CAS server.

```
libraries = CAS Get Libraries();
Show( libraries );
```

Export Data from a Data Table to a CAS Server

CAS Export Data() exports a data table to a CAS server to run an action on it.

```
CAS Export Data(
    Open( "$SAMPLE_DATA/Big Class.jmp" ),
    "casuser",
    "Big Class"
);
```

CAS Export Data() also has two arguments:

- Save(0) saves the data in memory.
- Save(1) saves the data to the file system.

Convert a CAS Table to a Data Table

CAS Table to Data Table() converts JSON data to a data table. A JSON element in the results of an action contains "_ctb". This is a marker indicating that it is formatted to represent a CAS table. You can send the JSON results to CAS Table To Data Table(). JMP will create a JMP data table from the JSON.

If there's more than one result, you might need to perform a little extra work as shown with Get Results at the end of the following example. This example shows how to export data, run an action, and save the data in several data tables.

```
cas = Current CAS Connection();
If( IsEmpty(cas),
    cas = CAS Connect( URL( url ), Username( username ), Prompt( "IfNeeded"
    ));
);
dt = Open( "$SAMPLE_DATA/Animals.jmp" );
cas << Export Data( dt, "CASUSER.Animals" );
```

```
json = "\[
{
  "table": {
   "name": "Animals",
   "caslib": "CASUSER"
   },
   "class": [
     "species",
     "subject",
     "season"
   ],
   "model": {
     "depVar": "miles",
     "effects": [
       {
         "vars": [
           "species",
           "season"
         ],
         "interaction": "BAR"
       }
     ],
     "printsol": true,
     "cl": false,
     "dfmethod": "RESIDUAL"
   },
   "random": [
     {
       "depVars": "miles",
       "effects": [
         {
           "vars": [
             "subject"
           ],
           "nest": [
             "species"
           ]
         }
       ]
     }
   ],
   "method": "REML"
}
]\";
action = New CAS Action(
  Action( "mixed.mixed" ),
```

```
   JSON( json )
);
rc = cas << Submit( action );
if(rc,
   results = action << Get Results;
   keys = results << Get Keys();
   For(i = 1, i <= N Items( keys ), i++,
      dt = CAS Table To Data Table( results[keys[i]] );
   );
);
cas << Disconnect();
```

`Get Results` enables you to examine the results of a CAS action; it's a JSL associative array.
`Get JSON` gives you the raw JSON; it's a JSL string.

The JSON data is in the format discussed at
https://developer.sas.com/apis/cas/rest/current/apidoc.html. Search for "ResultsTable".

In `CAS Table to Data Table()`, specify the JSON string, open the data table as invisible or
private, and control the table name and column names.

```
json = "jsonstring";
CAS Table to Data Table( json, <Invisible(Boolean)|Private(Boolean)>, <"Use
   Labels For Var Names">);
```

- `Invisible` hides the data table from view but shows it in the JMP Home Window.
  `private` hides the data table completely. Specify `Private` if the user doesn't need to
  interact with the data table.

- `"Use Labels for Var Names"` indicates that the table name and the column names will
  be set to the SAS dataset label and the column labels. The appropriate column property,
  SAS Name SAS Label, is used.

Note: CAS ResultsTable attribute values (peers to "_ctb") found in the "_ctb" JSON are also
set as table variables. The "CREATETIME" attribute creates a character table variable with a
locale-formatted timestamp (for example, 04/03/2019 4:38:05 PM).

Delete a CAS Table

`CAS Delete Table()` deletes the underlying file from the CAS file system.

```
CAS Delete Table( table name, <remove> ); // quote the table name
```

The `CAS Delete Table()` function has a remove option, which removes an in-memory table
from the CAS server. `CAS Remove Table()` serves the same purpose.

Remove a CAS Table

`Remove CAS Table()` removes an in-memory table from the CAS server.

```
CAS Remove Table( casuser, Big Class ) // quote both arguments
```

Verify the CAS Server Connection

`CAS Is Connected()` determines whether the CAS server is connected.

```
connected = CAS Is Connected();
Show( connected );
  connected = 1;
```

Get Details about the Current Connection

`Current CAS Connection()` obtains details about the current connection: the URL of the
CAS server, your user name, and the session ID.

```
connection = Current CAS Connection();
Show( connection );
```

Disconnect from the CAS Server

`CAS Disconnect()` disconnects from the current CAS server and optionally terminates the
session. By default, the session is terminated when disconnected.

```
CAS Disconnect(<"Terminate"|"NoTerminate">);
```

Terminate All CAS Sessions

`CAS Terminate Sessions()` ends all sessions owned by the current user except for the
current session.

```
CAS Teminate Sessions();
```

You can also specify the session ID to terminate the specified session.

```
CAS Terminate Sessions(session ID);
```

Use `Get Session()` to get the session ID.

```
url = "http://myCASserver"; // quoted string
cas = CAS Connect(
  URL( url ),
  Username( "my_username" ), // quoted string
  Prompt( "IfNeeded" )
```

```
    );
    session_id = cas << Get Session;
```

To terminate the current session, use CAS Disconnect().

Chapter **15**

Creating Projects

Organizing Project Files in JMP

A JMP project provides a way to organize files that you use in an analysis. You can add JMP files (reports, data tables, scripts, journals, and so on) and non JMP files, such as Microsoft Word or Adobe PDF files. After you perform an analysis, the data table, report, and graph appear in the project on tabs. You can display multiple reports or graphs, display the project or script editor log, and run scripts from the project. Reports and graphs remain linked to the data table.

Projects help you avoid the clutter of several open JMP windows. You can maximize the project window to have a bigger view of the project. And when you save the project, the state of the project is saved (for example, the open reports and the layout of the window).

Contents

Create a Simple Project

A JMP project provides a way to organize files that you use in an analysis and display the output. A simple project might display the data table, graphs, and reports on separate tabs in the project window.

```
project = New Project();
project << Run Script(
    dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
    dt << Run Script( "Bivariate" );
    dt << Run Script( "Distribution" );
);
```

Figure 15.1 shows the project with selected bars, showing that the graph is still linked to the data table.

Figure 15.1 Example of a Simple Project

Note: The best way to learn how to script a project is to create it interactively, change the
project file's extension to .jsl, and open the file in JMP.

Create a Project with a Bookmarks Pane and Log

A more complicated project might include a Bookmarks pane (to show file, folders, and
groups that use frequently), and a project log. The data table and output are in separate tabs.

Begin by creating a new project and creating a bookmark.

```
project = New Project();
project << Add Bookmarks({
            File( "$SAMPLE_DATA/Big Class.jmp" )
});
```

Run Script opens the data table and creates output inside the project.

```
project << Run Script(
  Open(
    "$SAMPLE_DATA/Big Class.jmp",
    Set Window ID( "data" ) // unique window ID
  );
  Distribution(
    Set Window ID( "distrib" ),
    Continuous Distribution( Column( :weight ) ),
    Nominal Distribution( Column( :age ) )
  );
  Graph Builder(
    Set Window ID( "graphbuilder" ),
    Size( 443, 409 ),
    Show Control Panel( 0 ),
    Fit to Window( "Maintain Aspect Ratio" ),
    Variables( X( :age ), Y( :height ), Y( :weight ) ),
    Elements( Position( 1, 1 ), Line( X, Y, Legend( 1 ) ) ),
    Elements( Position( 1, 2 ), Bar( X, Y, Legend( 2 ) ) )
  );
);
```

Set Layout specifies the layout of the tabs that show the data table and output.

```
project << Set Layout(
    V Splitter Box(
      Size( 1215, 700 ),
      <<Set Sizes( {0.77, 0.23} ),
      <<Dockable(),
      V Splitter Box(
```

```
              Size( 1215, 540 ),
              <<Set Sizes( {1} ),
              <<Dockable(),
              H Splitter Box(
                Size( 1215, 540 ),
                <<Set Sizes( {0.11, 0.42, 0.47} ),
                <<Dockable(),
                V Splitter Box(
                  Size( 128, 540 ),
                  <<Set Sizes( {0.33, 0.33, 0.33} ),
                  <<Dockable(),
                  Tab Page Box(
                    Title( "Window List" ), // show the Window List
                    Window ID( "Window List" )
                  ),
                  Tab Page Box(
                    Title( "Bookmarks" ), // show the Bookmarks pane
                    Window ID( "Bookmarks" )
                  ),
                  Tab Page Box(
                    Title( "Recent Files" ), // show the Recent Files pane
                    Window ID( "Recent Files" )
                  )
                ),
                Tab Page Box( // show the data table
                  Title( "Big Class" ),
                  Window ID(
                    "data"
                  )
                ),
                Tab Box(
                  <<Dockable(),
                  Tab Page Box( // show the Distribution graph and reports
                    Title( "Big Class - Distribution" ),
                    Window ID(
                      "distrib"
                    )
                  ),
                  Tab Page Box( // show the Graph Builder graph
                    Title( "Big Class - Graph Builder" ),
                    Window ID(
                      "graphbuilder"
                    )
                  )
                )
              )
            )
```

```
        ),
        Tab Page Box( Title( "Log" ), Window ID( "Log" ) )
    )
);
```

Save saves the project.

```
// if the project has already been saved, the Save message
// with no argument saves the file to the existing location
project << Save( "$DOCUMENTS/Big Class.jmpprj" );
```

Note: If you're saving a project with unsaved documents in JSL, the project doesn't close and you get a log message that the project has unsaved documents. All files in a project must be saved or closed before the project can be saved. However, if you interactively save a project with unsaved documents, you are prompted to save the documents.

Figure 15.2 Example of a Complex Project

Open the project from another script:

```
project = Open( "Big Class.jmpprj" );
```

In this example, the script that you are running to open the project is in the same folder as the project.

Running Scripts in a Project

When you run a script outside of a project, the reports generated by the script appear in different windows. When you run a script inside a project, the reports appear in different tabs inside the project.

Running a JSL script outside of a project

Open a Script Editor window by selecting File > New > Script. Paste and run the following script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Run Script( "Bivariate" );
dt << Run Script( "Distribution" );
Wait( 2 );
Get Window( "Big Class - Distribution" ) << Close Window();
```

Note that this script opens three JMP windows: one for the data table and one for each report. Close the reports and then the data table.

Running a JSL script inside of a project

Create a new project by selecting File > New > Project. From the project window, create a new script editor window by selecting File > New > Script. Note that the script editor window opens as a tab in the project. Paste the same JSL script you previously ran into the script editor and run it:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Run Script( "Bivariate" );
dt << Run Script( "Distribution" );
Wait( 2 );
Get Window( "Big Class - Distribution" ) << Close Window();
```

Note that this time, the data table and the reports all opened as tabs in the project, and that `Get Window()` correctly found the distribution tab. In general, all JSL functions that open, find, manipulate, and close JMP windows, tables, and reports work from within a project the same as they would outside projects.

Running a JSL script using a project's Run Script message

You can use the `Run Script` message to run JSL in the project's context. Open a Script Editor window, then paste and run the following code:

```
project = New Project();
project << Run Script(
    dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
    dt << Run Script( "Bivariate" );
```

```
dt << Run Script( "Distribution" );
Wait( 2 );
Get Window( "Big Class - Distribution" ) << Close Window();
);
```

Note that the script inside the Run Script message runs the same as if it had been in a script editor window inside the project.

Scoping Variables

JMP can have several projects open at the same time, as well as items not open in any project. Each project has its own global scope, which prevents conflicts caused by identically named variables in different projects. In the rare case that you do want to share specific variables among all opened projects, use the ::: root operator.

Create a Bookmark Group

You can create bookmark groups to organize the bookmarks in a project. A bookmark group can contain bookmarked files, folder, and additional bookmark groups. The bookmark group exists only in the project, not on your computer. Use Group() to create the bookmark group.

```
project = New Project();
project << Add Bookmarks(
        Group(
            "Sample Data",
            Expanded( 1 ), // open the group
            {File( "$SAMPLE_DATA/Air Traffic.jmp" ),
            File( "$SAMPLE_DATA/Big Class.jmp" )}
        )
);
project << Set Layout(
    H Splitter Box(
        <<Set Sizes( {0.15, 0.85} ),
        Tab Page Box( Window ID( "Bookmarks" ), )
    )
);
project << Save( "$DOCUMENTS/My Project.jmpprj" );
```

Figure 15.3 "Sample Data" Group in a Project

Close Tabs in Projects

Close a tab in a project by identifying the window with `Get Window()` and then closing it with `Close Window`.

```
project = New Project();
project << Run Script(
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
   dt << Run Script( "Bivariate" );
   dt << Run Script( "Distribution" );
);
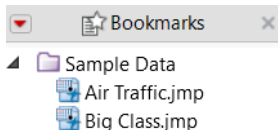
Wait( 2 ); // for demonstration purposes

// returns a reference to the open "Big Class - Distribution" window
dist = Get Window( Project( project ), "Big Class - Distribution" ) ;
dist << Close Window();
```

To close all tabs in a project, use `Get Window List << Close Window()`.

```
project = New Project();
project << Run Script(
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
   dt << Run Script( "Bivariate" );
   dt << Run Script( "Distribution" );
);

Wait( 2 ); // for demonstration purposes

Get Window List( Project( project ) ) << Close Window();
```

Get a Project

When running a JSL script from within a project, you can use the `This Project()` function to obtain that project. The following example displays the name of the current project in a new window.

```
project = This Project();
If( Is Empty( project ),

   /* if the script is not run from a project, display this
   sentence in the new window */
   New Window( "My Project", Text Box( "Script isn't running in a project." )
   ),
```

```
    /* if the script is run from a project, display the window title
    in the new window */
    New Window( "My Project", Text Box( project << Get Window Title() ) )
);
```

`Get Project()` takes the same arguments as the `Window()` function. Omit the arguments to get a list of the currently open projects, include a string to find a specific project by title, or include a number to get the *i*th open project.

```
Open( "$SAMPLE_PROJECTS/Big Class.jmpprj" );
Open( "$SAMPLE_PROJECTS/Sports.jmpprj" );

// print the title of Sports.jmpprj
Print( Get Project( 2 ) << Get Window Title() );
```

`Get Project()` is also helpful when you want to save a project or set the layout from within the project.

```
Get Project() << Save(...);
Get Project() << Set Layout(...);
```

Reset the Window Layout

Suppose that you create a project and customize the layout of the windows. For example, you might hide the Window List. Use the `Reset Layout` message to return the windows to their original layout.

```
project = New Project();
project << Run Script( // open data table and create a graph
    Open(
        "$SAMPLE_DATA/Big Class.jmp",
        Set Window ID( "data" )
    );
    New Window( "Big Class - Bivariate of weight by height",
        Set Window ID( "bivariate" ),
        Bivariate(Y( :weight ), X( :height ),
        )
    ) << Set Window Icon( "Bivariate" );
);

Wait(3); // for demonstration purposes
project << Set Layout( // set a custom layout
    H Splitter Box(
        <<Set Sizes( {0.2, 0.8} ),
        Tab Page Box( Window ID( "Bookmarks" ) ),
```

```
                    V Splitter Box(
                        <<Set Sizes( {0.53, 0.47} ),
                        Tab Page Box(
                            Window ID( "bivariate" ) ),
                        Tab Page Box(
                            Window ID( "data" ) )
                    )
            )
    );

    Wait( 3 ); // for demonstration purposes
    project << Reset Layout(); // return to the original layout
```

Run Startup Scripts to Control Workspaces

A workspace is either a project window or the windows that are not in a project (for example, the home window). A default workspace loads when JMP starts, and then each time you create or open a project, JMP creates another workspace.

Consider writing a workspace startup script to customize variables for a specific project. You might also construct a new customized window in a workspace startup script to put a copy of that window in each project. This is an alternative to having a single copy of the script that runs outside of projects.

The workspace startup script runs each time a new workspace loads. This means that the workspace startup script runs once when JMP starts (when the "default workspace" loads) and then runs again each time a new project is loaded. The workspace startups script runs in the context of the project, so any windows it opens or variables it sets are for that project only.

Name the script workspaceStart.jsl and place it in one of the following folders, as appropriate for your operating system. When JMP starts, JMP looks for the workspaceStart.jsl script in these folders in the order in which they are listed here. The first one that is found is run, and the search immediately stops.

Note: Some path names in this section refer to the "JMP" folder. On Windows, in JMP Pro, the "JMP" folder is named "JMPPro".

On Windows:

1. C:/Users/<username>/AppData/Roaming/SAS/JMP/16

2. C:/Users/<username>/AppData/Roaming/SAS/JMP

On macOS:

1. /Users/<username>/Library/Application Support/JMP/16

2. /Users/<username>/Library/Application Support/JMP

The workspaceStart.jsl script runs only for a particular user on a computer. You can add a script named workspaceStartAdmin.jsl in one of the following places, as appropriate for your operating system. This script is run for every user on a computer. JMP runs workspaceStartAdmin.jsl first if found. Then JMP runs workspaceStart.jsl if found.

On Windows:

1. C:/ProgramData/SAS/JMP/16
2. C:/ProgramData/SAS/JMP

On macOS:

1. /Library/Application Support/JMP/16
2. /Library/Application Support/JMP

Creating Applications

Application Builder

In addition to providing platforms for quickly analyzing data, JMP lets you create applications to support your specific needs. You often perform the same tasks every day (such as running Distribution and Fit Model analyses on a data table and viewing the results). In JMP's Application Builder, you can create an application that shows the results for both analyses in one window. Application Builder's drag-and-drop interface reduces the amount of scripting required to create an application.

Add-In Builder visually guides you through creating an add-in that users can install in JMP. And after creating an application in Application Builder, use Add-In Builder to save the application as an add-in. This allows users to install the application and quickly run it inside JMP.

Figure 16.1 Creating Customized Reports in Application Builder

Contents

Create Applications with Application Builder

Application Builder's drag-and-drop interface lets you visually design windows with buttons, lists, graphs, and other objects. This saves you the step of writing scripts to create these objects.

Application Builder also helps boost the productivity of experienced JMP Scripting Language (JSL) programmers. You can start designing a program in Application Builder and edit the automatically generated scripts. Integrate your own scripts to create even more powerful custom applications.

You can use Add-In Builder with Application Builder. First, use Application Builder to write complex scripts that create new platforms and interact with users and JMP. Then, use Add-In Builder to easily distribute your complex scripts so that any JMP user can access them without having to open and run the scripts. See "Compile Add-Ins with the JMP Add-In Builder" on page 889

Notes:

- Creating simple applications in Application Builder requires little to no JSL programming. However, to create most simple applications, Dashboard Builder might be sufficient. Dashboard Builder provides a subset of features that are found in Application Builder. See *Using JMP*.

- In each version of JMP, Application Builder takes advantages of new features that are not supported in earlier releases. For maximum compatibility, we recommend that you create applications in the earliest version of JMP that the application will be run in.

Example of Creating an Application

Application Builder helps you create instant applications without writing JSL. One example is a JMP window in which you arrange reports, graphs, and other objects to make them easy to interact with at once.

This example shows how to create a simple instant application for a Distribution report. This application lets you analyze preselected variables rather than having to select them in a launch window.

1. Select **Help > Sample Data Library** and open Big Class.jmp.
2. Run the **Distribution** table script to generate the report.
3. Select **File > New > Application** (or **File > New > New Application** on macOS).
4. Click the Blank Application box.

   A new blank application appears. Notice the Distribution report in the left column.

Figure 16.2 New Application with Distribution Report in the Sources Panel

5. In the Sources panel, drag the Distribution report onto the application workspace (the blank area on the Module1 tab).

6. Click the Application Builder red triangle and select **Run Application**.

    The Distribution report appears in a new window. Note that the histograms are interactive and still associated with the data table. When the data changes, you run the application again, and the Distribution report is updated.

Figure 16.3 Example of an Instant Application

Application Builder Terminology

An *application* consists of modules. A *module* contains the objects and scripts that are compiled into an application. When you run an application, JMP can show a separate window for each module. For example, one window might show a **Create Graphs** button. After the user clicks the button, the graphs appear in a new window.

There are terms you should become familiar with before using Application Builder. Knowing these terms helps you understand how a module works, even if you need to minimally edit an automatically generated script.

Application   The top-level file that consists of one or more modules.

Module   A collection of objects, messages, instances, and other JSL statements that are compiled into an application to create new JMP features.

Instant Application   An application that typically consists of reports and does not require JSL scripting. You can also create an instant application, or "dashboard", in Dashboard

Builder. Similar to Application Builder, Dashboard Builder has a drag-and-drop interface but with pre-configured display boxes.

Custom Application   An application that demonstrates custom behavior through JSL scripting.

Objects and Messages   An *object* is a dynamic entity in JMP, such as a data table, a data column, a platform results window, a graph, and so on. Most objects can receive messages that instruct the object to perform some action on itself.

A *message* is a JSL statement that is directed to an object. That object knows how to evaluate the message.

Module Instance   The occurrence of a module in the application. In a complex application, you might create a start-up script that creates multiple instances of one or more modules.

Namespace   A collection of unique variable names and corresponding values that prevents collisions among scripts.

JMP creates the `Application` namespace and the `ModuleInstance` namespace automatically. Symbols in the `Application` namespace are visible only to scripts in the application; the namespace is not available to scripts not created in the application. For more information about namespaces, see "Advanced Scoping and Namespaces" on page 272 in the "Programming Methods" chapter.

Variable   A unique name to which you assign a value. JMP creates the `thisApplication` and `thisModuleInstance` variables automatically.

- The `thisApplication` variable (used in the `Application` namespace) contains module names, which are available in any script in the application.

- The `thisModuleInstance` variable (used in the `ModuleInstance` namespace) contains box and script names, which are valid only in their own modules.

Container   A display box object (such as a tab or outline) into which you drag other objects.

Design an Application

Before creating the application, you must identify its purpose, decide on which graphical elements to include, and figure out which JSL scripts you need to write.

Purpose   What do you want the application to accomplish? Which features of JMP do you need to customize (for example, a launch window, report, or graph)?

Graphical Elements   How many modules do you need? Look at the list of graphical objects in Application Builder and decide which objects to include in your application. The Sources panel shows dozens of boxes and icons that you can drag and drop into a module. Also look at the preinstalled samples to see whether a similar module exists.

Scripting   Depending on the complexity of the application, you write JSL scripts. See "Write Scripts" on page 868 for more information about writing scripts for an application.

A non-interactive application that only shows objects does not require scripts. One example is a window showing reports that you have already generated in JMP.

Application Builder Window

Figure 16.4 shows an application under development in Application Builder.

Figure 16.4 The Application Builder Window

The Application Builder window provides the following features:

• The toolbar provides quick access to many features, such as aligning objects and inserting common display boxes. (Select **View > Toolbars > Application Builder** to show the toolbar.)

- The Sources panel shows objects that you can include in the application, including open reports and graphs. Right-click an object and select **Scripting Help** for information about the object and its properties.

- The workspace in the middle shows dotted grid lines, which help you line up objects. Drag objects onto one or more Module tabs in the workspace and write scripts on the Scripts tab to control the objects' functionality.

- The Objects panel shows a hierarchical view of the application. Along with the application, each module and its nested objects are listed. You can click an object to show its properties or to select it on the workspace.

- In the Properties panel, you set properties for each object, such as the location, width, or name of an object. The properties vary depending on the type of object.

Tip: To hide the grid and turn off snap-to-grid for all applications that you create, select **File > Preferences > Platforms > JMP App** and deselect these options. You can also deselect these features in the Application Builder red triangle menu.

Application Builder Red Triangle Options

The Application Builder red triangle menu provides options for running and debugging the application, opening sample applications, showing the grid, and more.

Run Application   Starts the application. A window for each module opens, and you can interact with the application as the user will.

Debug Application   Opens the script in the JSL Debugger to troubleshoot errors. See "Debug or Profile Scripts" on page 77 in the "Scripting Tools" chapter.

Snap to Grid   Aligns (or snaps) objects to the nearest dotted grid lines as you drag them on the workspace. Selected by default.

Show Grid   Displays dotted grid lines on the workspace. Selected by default.

Show Properties   Shows or hides the Objects and Properties panels.

Auto Scroll   Automatically scrolls horizontally or vertically as you drag an object near the edges of the workspace. Selected by default.

Dashboard Mode   Displays the application in the simpler Dashboard Builder window. When you drag a report, data filter, or data table to the application, it is automatically placed in splitter and tab page boxes. You cannot add scripts in Dashboard Mode.

Be aware that switching back and forth does not change anything that you have already added to the application. Switching modes changes the way the interface works when you add new content.

Source Panel   Shows or hides the Sources panel (the left column in which application components are displayed) and categorizes the icons. Both settings are selected by default.

Save Script   Lets you save the application to a data table, journal, script window, template, or add-in. See "Options for Saving Applications" on page 872.

Table 16.1 Sample Applications Installed with JMP

Six Quality Graphs.jmpappsource	Creates three Control Charts, a Distribution report, and a Capability Analysis report.
Data Filter Compare.jmpappsource	Creates two Bubble Plots. Each plot has its own data filter and Tabulate report. See this application for an example of the `Data Filter Context Box` function.
Data Table Application.jmpappsource	Lets the user select data table columns to stratify and the sampling rate. Display the data table to see the results.
Graph Launcher.jmpappsource	Lets the user enter an equation and axes settings and then create a graph. The initial window remains open, so the user can modify the equation and create new graphs.
Instant App.jmpappsource	Combines two Multivariate reports (Principal Components/Factor Analysis and T Square with All Principal Components).
Instant App Customized.jmpappsource	A modified version of Instant App.jmpappsource. Options for selecting the principal components report, changing the marker size, and showing means are included.
Launcher With Report.jmpappsource	Lets the user select data table columns in a launch window and then creates a graph.
Parameterized Instant App.jmpappsource	Lets the user select data table columns and then creates two Multivariate reports. An argument is assigned to the Y role. This means that the reports can be created from any open data table, not just the table specified in the application.
Parameterized MSA Combo Chart.jmpappsource	Creates a collection of reports for Measurement System Analysis (MSA).

Table 16.1 Sample Applications Installed with JMP *(Continued)*

Presentation.jmpappsource	Creates an onscreen presentation, similar to a slideshow, with navigation buttons and an embedded script.
R Application.jmpappsource	Lets the user select columns and then shows multivariate data in the shape of a face (the Chernoff faces). Requires the R TeachingDemos package.
SAS Application.jmpappsource	Runs a SAS script and then adds the output to a report. Prompts you to log on to a SAS server if you are not already connected.

Create an Application

After writing the application specifications, you create a blank application in JMP and begin adding objects and scripts.

This section provides the basic steps for creating an application:

- "Create a New Application" on page 860
- "Arrange and Remove Objects" on page 863
- "Customize Object Properties" on page 866
- "Write Scripts" on page 868

Note: In each version of JMP, Application Builder takes advantages of new features that are not supported in earlier releases. For maximum compatibility, we recommend that you create applications in the earliest version of JMP that the application will be run in.

Create a New Application

To create a new blank application, follow these steps:

1. Select **File > New > Application** (or **File > New > New Application** on macOS).
2. Select the Blank Application.

   The Application Builder window appears.
3. Select **File > Save** and save the file as a JMP Source File. The extension is .jmpappsource.

After you create the blank application, you add open reports, display boxes, and other objects to the workspace. See "Example of Creating an Application" on page 853.

Manage Modules

When you create a new application, a Module tab appears by default. You can add objects to this module to create interface items. To create additional windows for your application, you can add and customize new modules.

Add Objects to a Module

Follow these steps to add objects to a module:

1. In the Sources panel, select the type of object that you want to add.

2. Drag the object onto the **Module1** tab (or double-click the object).

3. Select the object and update its properties in the Properties panel. See "Customize Object Properties" on page 866.

4. Add scripts to scriptable objects. See "Write Scripts" on page 868.

5. (Optional) To prevent a module from launching when you run the application, select the module in the Objects panel and deselect **Auto Launch**. (You might do this while testing one out of multiple modules.)

6. Click the Application Builder red triangle and select **Run Application** to test your application.

   Depending on how the modules are used, it may not be possible to test one module without creating the other modules.

Create a Module

1. Select **Format > Add Module.**

2. In the Properties panel, select the **Module Type**:

   – Dialog

   – Dialog with Menu

   – Modal Dialog

   – Launcher

   – Report

   – Display Box

Tip: When you run an application, the window title is the name of the data table, if applicable, followed by a hyphen and the application name. Change the application name by modifying text in the JMP App object's **Name** property.

Rename a Module

Change the **Variable Name** in the module's object properties.

Delete a Module

Select **Format > Delete Module.**

Delete a Data Table

Select the Data Table object, right-click and select **Delete.**

Note: You cannot delete the data table if it has any associated objects. Delete any objects and then delete the data table.

Modal Dialog Modules

Modal Dialog modules have some special behaviors to be aware of when using them.

- `ret = Module1 << Create Instance()` will not return until the dialog is complete. By this time, the dialog has been destroyed. Therefore, the return value will not be a handle to the Module Instance. Like `New Window()`, the return will instead be of the form `{Button(1 | -1), User Data}`. (1 | -1) indicates whether the OK (1) or Cancel (-1) button was pressed.

- `User Data()` is a new property on the Module Instance. During execution of the dialog, the Module script can set the user data:

      thisModuleInstance << Set User Data(...);

  to store something to be parsed by the caller. The user data will be evaluated at the time that it is set. There is also a corresponding `Get User Data()`.

- Like `New Window()`, if there is no OK or Cancel button included, an OK button will be added.

- There is an optional `Validate` script property on the Module Instance. This script is only used for Modal dialogs and behaves like `On Validate()` in `New Window()`. It is called when the OK button is pressed, and returns 1 to accept the input or 0 to disallow the close.

Tip: If you use the Validate property, you can call `thisModuleInstance << Set User Data()`.

- Unlike other module types, a Modal Dialog will not show until the module script has completed. For other module types, the window is created during the call to `thisModuleInstance << Create Objects`. JMP cannot display a Modal Dialog here because control would stop and there would be no way to initialize the contents of the boxes in the display. Since the window is not created yet, you will not be able to do actions like set the window title or set an on-close script.

Arrange and Remove Objects

Drag objects from the Sources panel onto the workspace or double-click them. A blue border appears around the edge of a selected object. Then you can position the selected object in several ways. You can also change the container of an object or insert an object into a new container.

Tip: To quickly select an object, select it in the Objects panel. This method is particularly helpful when the internal object completely covers the container, as with a border that has an internal text box.

Drag an Object

One way to position objects is by dragging them around the workspace.

- The top and left edges of an object align (or snap) to the nearest dotted grid lines as you drag them on the workspace. For a more precise placement, click the Application Builder red triangle and deselect **Snap to Grid**. You can also turn off **Snap to Grid** for all applications in the JMP Preferences. Select **File (Windows) or JMP (macOS)> Preferences > Platforms > JMP App** to change the option.

- Position objects in containers instead of positioning them independently on the workspace. For example, place two button boxes in a V List Box to align them vertically.

- When you drag an object, press Shift to restrict movement to a single axis (for example, to move the object horizontally without moving the vertical position).

- When you drag an object into a container, the arrow indicates where you can drop the object.

- When you drag an object over another one, a blue line shows where you can drop the object.

Figure 16.5 shows the various methods for dragging objects inside a container.

Figure 16.5 Examples of Dragging Objects into a Container

Change the X and Y Positions on a Container Object

- To position a container object precisely, select the object and then change the **X Position** and **Y Position** properties. After you enter a new X position, press Tab to see the object move to the new position and then enter the new Y position.

- To place an object in the upper left corner, right-click the object and select **Move to Corner**. This is a shortcut to setting the X and Y positions to 0. On macOS, press Ctrl+Command and then select **Move to Corner**.

Line Up Multiple Objects

To line up objects horizontally or vertically, select the objects and then select an option from the **Format > Align Boxes** menu.

Tips:

- The **Align Boxes** options are unavailable when you select a container rather than the internal objects.

- If you right-click a container, you might inadvertently select one of the internal objects instead. Right-click in the workspace instead.

Change the Type of Container

After inserting some containers, you can change them to a different type of container without having to re-create the object. Suppose that you create a panel that contains text and a button. To see what the panel looks like as an outline, select the panel, right-click in the module, and then select **Change Container > Outline Box**.

Figure 16.6 Change a Panel to an Outline

In this example, you would also change the outline title to match that of the panel.

Tip: To change the orientation of a list box quickly, select or deselect **Horizontal** in the object properties.

Insert Objects into a New Container

The toolbar at the top of the Application Builder window provides buttons for containers such as border boxes and mouse boxes (Figure 16.7).

To insert selected objects into a container, click the appropriate button on the toolbar or select **Format > Add Container**.

Figure 16.7 Container Toolbar

Duplicate Objects

When you copy and paste an object, the second instance of the object is renamed. The scripts attached to the object are also renamed. Pressing Ctrl (or Command on macOS) and dragging the object also creates duplicates of the originals.

Delete an Object

- Remove one or more objects by selecting them and then pressing Delete.
- Drag the objects outside of the module.
- On macOS, select the objects, press Ctrl+Command, and then select **Delete**.

- On Windows, select **Edit > Clear** to delete all objects in a module.

- Select the object and then press Backspace.

If you no longer need the object's script, you can delete the script also.

Customize Object Properties

After you drag an object from the Sources panel onto a module, customize the object properties in the Properties panel. Editing the object properties saves you the step of writing JSL for list items, button names, and so on.

Variable names are case- and space-insensitive. When a script contains an object named "Button1", Application Builder warns if you try to rename another object "Button 1".

To create the radio box list items in Figure 16.8, double-click the placeholder items "Item1" and "Item2" and enter the new list item.

Figure 16.8 Radio Box Object Properties

For more information about an object, right-click the object and select **Scripting Help**. The JMP Scripting Index appears, and the object that you selected is highlighted. The Scripting Index often includes a script that you can run to see an example of the object.

To read more about values in the Properties panel, hover over the value.

Application Properties

The Application properties identify the run password, data table name, and more. Select the application in the Objects list.

Name　Appears after the data table name in the application's title bar.

Description　Enter a description of the application.

Auto Launch　Shows a launch window in which the user selects the arguments that have been defined in the application. This launch window is not one of the defined modules; the modules will be instanced based on their own Auto Launch property.

Encrypt　Prevents users from editing the application in a text editor. Only applications that the user runs are encrypted (the .jmpapp file and the application in a JMP add-in). Scripts that you save to a data table, journal, and add-in are encrypted. For more information about encryption, see "Encrypt and Decrypt Scripts" on page 304 in the "Programming Methods" chapter.

Run Password　Enter the password required to run the application. To test the password, run the application outside Application Builder.

Table Module Properties

Inserting the Data Table object creates a data table object. To define properties such as the data table path, select the data table module in the Objects panel. Then modify the following options in the Properties panel.

Variable Name　Specifies the name of the data table object. This name appears in the Properties and Objects panels and in the application's JSL script.

Path　Specifies the absolute or relative filepath for the data table used in the application. You can precede the data table name with a path variable (such as $HOME or $USER_APPDATA).

Application Builder opens the specified data table when you edit the application. If the Path property is empty or the data table cannot be found, you are prompted to open the table.

When you close a data table and objects in the application depend on the data table, the objects are removed from the application and a warning message appears. To restore the objects, reopen the application.

Label　Specifies the string used when prompting the user to open a data table. Figure 16.9 shows the default value.

Figure 16.9 Label in Data Table Prompt Window

Location   Determines how the data table used in the application is selected when the user
   runs the application.

– Current Data Table: Uses the current data table. If no data tables are open, the user is
   prompted to open one. This option gives users the flexibility to use their own data.

– Full Path: Uses the data table specified in the Path property.

– Name: Uses the first open data table with the specified name. Otherwise, JMP uses the
   data table specified in the Path property. Note that if the application is moved to
   another computer, the link to the data table is broken. Consider storing the data table in
   a shared location.

– Prompt: Asks the user to select an open data table or browse to select a data table.

– Script: Uses the data table defined in the application or module script.

– Embedded Script: Reads the Source script in the data table that was used to generate
   the report. The Source script is run when you open the application.

Invisible   Hides the data table from view but lists it in the JMP home window. This option is
   available for the Full Path and Name locations.

Write Scripts

After adding an object to the module, you write a script for the object to provide functionality.
For example, you might want the user to click a button, select a directory, and then select a
data table. Or you might have an application that displays a different graph based on which
radio button the user selects.

Application and Module Namespaces

To prevent variable names and values from conflicting among scripts, Application Builder
automatically creates the Application namespace and a namespace for each module:

• In the Application namespace, you write scripts that are executed when the user runs the
   application. Functions defined in the Application namespace can be used in any module.

- In the Module namespaces, you write scripts that are run when an instance of that module is created. If clicking a button in the application opens a new launch window, that launch window is an instance of the module. Two instances of the same module have their own copies of any variables or functions.

To see the scripts in these namespaces, click the **Scripts** tab and then select the namespace in the Namespace list (or in the Objects panel).

Figure 16.10 Application and Module Namespaces

There are two types of scripts: named scripts and anonymous scripts.

Write a Named Script

A *named script* is a function that several controls can use. The `this` argument tells which control is calling the function. In the following example, `Get Button Name` is sent to the `this` argument to print the button name to the log when the button is clicked:

```
Button1Press = Function({this}, Print(this <<Get Button Name))
```

On another button, use `Button1Press` script to produce the same results.

Two other advantages of named scripts: when you add a named script to an object, JMP inserts a placeholder script on the Scripts tab, which you then edit. On the Scripts tab, you can also select a script from the Method list to navigate to that code, which is particularly helpful in long scripts.

Add a named script:

1. Right-click the object, select **Scripts**, and then select the script that you want to add. In this example of creating a button, select the **Press** script. (On macOS, press Control, and then select **Scripts > Press**.)

The object's placeholder script appears on the **Scripts** tab (Figure 16.11). The name of the named script, *Button1Press*, shows up in both the script and the object properties.

Figure 16.11 New Script and Script Properties

```
Button1Press=Function({this},
    // This function is called when the button is pressed
    name = this << Get Button Name;
);
```

Button Box	
Variable Name	Button1
Position	X 40 Y 0
Title	Submit
Style	Normal
Pressed	☐
Icon	
Icon Location	Right
Press	... Button1Press

2. Edit the object's placeholder script and properties to provide the necessary functionality. For example, the button box shown in Figure 16.11 was renamed. The Title property was changed to *Submit*, the text that appears on the button.

Tip: After you add a script to an object and then delete the object, delete the object's script from the **Scripts** tab if you no longer need the script. This feature prevents scripts that you might want in the future from being deleted.

If you rename the script in the object properties, rename it on the **Scripts** tab also. And if the script is used in another part of the application, rename it there as well.

Write an Anonymous Script

An *anonymous script* is available only to the object that defines it. For example, you might want a simple `Print` statement for one button that is not used elsewhere. By writing an anonymous script, you reduce the number of names to manage in the script. Anonymous scripts also reduce clutter among more important named scripts in the Scripts tab, because you add them to the object properties.

The following examples show two types of anonymous scripts:

```
Print(Button1 <<GetButtonName) // simple anonymous
Function({this}, Print(this <<GetButtonName)) // parameterized anonymous
```

Notice that the simple script sends a message to the "Button1" variable, but `this` is the control in the parameterized script.

Objects such as check boxes might provide additional arguments after `this` that are not otherwise available. One example is an argument that tells which check box in a column of check boxes just changed.

Write an anonymous script:

1. Select the object and click ⬚ in the object properties.

    The anonymous script editor appears.

2. Enter the script and click **OK**.

   The text of the anonymous script appears in the object's properties (rather than the name of a named script).

Tip: Avoid copying and pasting anonymous functions to simplify code maintenance; use a named script instead if the script is needed in more than one place.

Show Specific Scripts

There are several ways to view a script for a specific object:

• Double-click the object on the **Module** tab.

• When an object has multiple scripts, right-click the object, select **Scripts**, and then select the script name. You can also select the **Scripts** tab, select the module name from the **Namespace** list and then select the script name from the **Method** list. Likewise, to see the application scripts, select **Application** from the **Namespace** list.

In each case, the **Scripts** tab appears with the cursor in the first line of the object's script.

Tip: To make a poorly formatted script easier to read, right-click and select **Reformat Script**.

Copy and Paste Objects with Scripts

When you copy and paste an object that has a script, the second instance of the object and script are renamed.

Edit a Platform Script

You can select the object for a platform in an application and edit the script for that platform. For example, you might edit the script for the Multivariate platform to show the scatterplot matrix or to use different columns.

Right-click the platform object on the workspace and select **Edit Platform Script**. After you edit the script and click OK, the object on the workspace is updated.

Edit or Run an Application

To open an application for editing, select **File > Open**, select the .jmpappsource file, and then select **Open**.

Note: If the Table property is empty or the specified data table cannot be found, the application runs anyway but objects requiring the data table will fail to create and JMP displays a warning.

To run an open application, click the Application Builder red triangle and select **Run Application**.

To run a closed application, select **File > Open**, select the .jmpapp file, and then select **Open**.

On Windows, you can also open or run an application from the JMP Home Window by doing one of the following:

- Drag the file from Windows Explorer onto the JMP Home Window or onto a blank application window.

- Double-click the file in Recent Files list.

- Right-click either the .jmpappsource or the .jmpapp file and select either **Edit Application** or **Run Application**.

Options for Saving Applications

JMP provides several options for saving application files.

- When you select **File > Save As** on Windows, you can save the application as an application source file (.jmpappsource), an application (.jmpapp), or a script.

- Selecting **File > Export** on macOS enables you to save the application as an application file or a script. To save the application as an application source file on macOS, select **File > Save As**.

The .jmpappsource file allows an application developer to save the application as is, in order to continue editing the application later. When reopened, the App Builder interface remains in same state as when it was saved. The data tables used to create the application must still be available.

The .jmpapp file is used to distribute applications for others to use. The application runs immediately after opening the file. The application resolves the data tables using the current data tables, files on disk, or by prompting the user, depending on the application settings.

The Save Script red triangle menu provides additional options for saving scripts. When an encrypted script is saved, JMP encloses the script in the `JSL Encrypted()` function to preserve white space and comments.

To Data Table   Saves the script to the data table that was used to produce the report. This enables you to run the script again from the data table to recreate the results. When you edit this script from the data table, the application opens in Application Builder, not the script editor.

To Journal   Saves a button that runs the script in a journal. The script is added to the current journal. The script contains the path to the data table. Note that if the data table cannot be found, the script does not run.

To Script Window   Opens a script editor window and adds the script to it. If you have already saved a script to a script window, additional scripts are added to the bottom of the same script window. Note that if you modify the application significantly in a script window, you might not be able to edit the application in Application Builder. This option is useful when you want to embed the application in a larger JSL process.

Tip: This can also be accomplished by saving as a .jmpapp file and using `Open()` to run the application. However, saving the script can reduce the number of files.

To Add-In   Lets JMP users install the application and launch it from a JMP menu. See "Compile Add-Ins with the JMP Add-In Builder" on page 889.

To Template   Saves the application as a template from which you can create new applications. The templates are saved in the following locations based on your operating system and JMP version:

– Windows (JMP): "/C:/Users/<username>/AppData/Roaming/SAS/JMP/<version number>/ApplicationTemplates"

– Windows (JMP Pro): "/C:/Users/<username>/AppData/Roaming/SAS/JMPPro/<version number>/ApplicationTemplates"

– macOS: "/Users/<username>/Library/Application Support/JMP/<version number>/ApplicationTemplates"

Additional Examples of Creating Applications

The following examples illustrate various uses for applications in JMP.

Parameterizing Variables

The following example shows how to create an application with parameterized variables. Users select the variables in a launch window, and then predefined reports are generated.

1. Select **Help > Sample Data Library** and open Quality Control/Steam Turbine Historical.jmp.

2. Run the Principal Component Analysis and Loading plot table scripts to generate the reports.

3. Select **File > New > Application**.

4. Select **Blank Application**.

   The Application Builder window appears.

5. Enlarge the window.

6. In the Sources panel, drag each Multivariate report onto the application workspace in a single row.

7. Select **Edit > Select All**.

8. Select **Format > Add Container > H List Box** to align the reports horizontally.

9. Select each report and type yvar next to **Y Variable** in the Objects panel.

10. Click the Application red triangle and select **Run Application**.

11. Select the Fuel, Steam Flow, Steam Temp, and MW variables and then click the **Y** button.

12. Click **OK**.

New Multivariate reports appear in one window.

Figure 16.12 Multivariate Application

Tip: The absolute path to the data table is inserted automatically in the application's Table property. If the data is sample data, the $SAMPLE_DATA path variable is automatically used in the path. You can also enter an absolute or relative path. Remember that the user must have access to this path.

Filtering Data in Multiple Reports

In an application that contains several reports, you can select data in one report and then view only that data in all other reports contained in the same window.

To set up the filter, follow these steps:

1. Create the application and add two reports to the workspace.

2. Select both reports. From the toolbar, select the HListBox icon.

3. Right-click the primary report and select **Use as Selection Filter**.

   This places the primary report display box in a Data Filter Source Box and the parent report display box in a Data Filter Content Box.

4. Click the Application Builder red triangle and select **Run Application**.

   The reports appear in one window (Figure 16.13).

5. Test the application by selecting a histogram bar in the primary report.

   Only data for the selected bar appears in the second report.

Figure 16.13 Example of Filtered Content

Filtering Data in Separate Reports Using a Local Data Filter

The following example illustrates how to add a Local Data Filter to multiple reports and then filter each report separately.

1. Select **Help** > **Sample Data Library** and open SATByYear.jmp.

2. Run these scripts: Bubble Plot by Region and Graph Builder Map.

3. Resize the report windows to make them smaller, so that a 2x2 arrangement will fit on the screen.

4. Select **File** > **New** > **Application** and choose **Blank Application**.

5. Double-click **Data Filter Context Box**.

    This adds a data filter context box to the workspace. Everything placed in this box will
    share the same Local Data Filter.

Figure 16.14 Adding a Data Filter Context Box

6. Under Containers, drag a **H List Box** into the Data Filter Context Box.

    This allows you to put multiple objects in this box and arrange them horizontally.

Figure 16.15 Adding a H List Box

7.   Now drag a Data Filter (local), the Bubble Plot, and the Tabulate report into the H List Box.

Figure 16.16 Adding Local Data Filter and Two Reports

8. Select the entire Data Filter Context Box. Press Ctrl and drag it below itself, to place a copy below the original set of reports.

9. Select both Data Filter Context boxes, then on the toolbar, click the V List Box icon.

    This arranges both reports into a single list box for display purposes.

Figure 16.17 Selecting Both Data Filter Context Boxes and the V List Box Icon

10. Click the Application Builder red triangle and select **Run Application**.

11. In the Local Data Filter on the top, click SAT Math, then click the Add button ⊞ . Move the slider to see how it only impacts the reports at top. Do the same for the bottom set of reports.

Selecting Dates

To insert a date selector window into an application, follow these steps:

1. Drag a Number Edit Box from the Sources panel to the workspace.

2. Select the Number Edit Box and then click the button next to Format in the Properties panel.

3. Select **Date** from the list and then select **m/d/y**.

4. Click outside the list.

5. Click the Application Builder red triangle and select **Run Application**.

   The Number Edit Box, which shows a date, appears in a new window.

6. Click the calendar icon 🗓 to view the date selector window.

Figure 16.18 Example of a Date Selector

The date selector enables you to select the month and year as well as the date and time for the box.

Creating a Launch Window and Report

The Launcher with Report sample application combines a platform launch window with the resulting report. The sample application is displayed when you select **File > New > Application**.

The following example shows you how to arrange objects and include scripts that make up the application. You will recreate the Launcher with Report sample application so that you can create your own applications similarly.

Note: In the Launcher with Report sample application, the variable names of display boxes are customized to describe the display box. For example, "Text1" was renamed "Description" in the sample application. To reduce the number of steps in this example, you do not rename the variables unless one of the scripts refers to the variable name.

---

1. Select **File > New > Application** and click the **Launcher with Report** sample.

   You will copy and paste JSL from this sample into your copy. Notice that the Iris.jmp sample data table opens when you open the application.

2. Select **File > New > Application** and click **Blank Application**.

Create the Launch Module

The Launch Module contains objects that create a launch window.

1. In the Objects panel of the blank application, select Application.

2. In the Properties panel:
   − Change the Name from Application to "Radviz" and press Enter.

      This name appears in the title bar when you run the application.

   − Change the Description to "Launch Window with Report".

Figure 16.19 Customizing the Application

3. Click the **Scripts** tab and select Radviz from the Namespace list.

4. After the green comments, on a new line, type the following expression:

   `dt = Current Data Table();`

   One of the scripts that you add to this application refers to `dt` to identify the data table.

5. In the Objects panel, select **Module1**.

6. In the Properties panel:
   − Change the Variable Name to "LaunchModule" and press Enter.

   − Change the Module Type to Launcher.

   − Make sure that Auto Launch is selected so that the launch window opens when you run the application.

Figure 16.20 Customizing the LaunchModule

7. Click the **LaunchModule** tab and drag a V List Box (under Containers) onto the workspace.

8. Drag a Text Box (under Display) over the arrow of the V List Box.

9. Double-click Text1, change it to "Radviz", and then press Enter.

10. Drag an H List Box (under Containers) below the border of the Text Box until a blue line appears.

Figure 16.21 Adding the H List Box

11. Because you have made many changes, select **File > Save** and rename the file. Make sure that .jmpappsource is selected as the file type.

Create the Select Columns Box

1. Drag a Panel Box (under Containers) over the H List Box arrow.

2. In the Properties panel, change the Title to "Select Columns" and press Enter.

3. Drag a Col List Box(All) (under Input) over the Panel Box arrow.

4. In the Properties panel, change the Variable Name to "ColumnList" and press Enter.

    The script you add later refers to this variable name.

5. Select **DataTable1** in the Objects panel.

    The current data table, Iris.jmp, appears in the Path box.

Create the Cast Selected Columns into Roles Box

1. Drag a Panel Box (under Containers) to the right of the existing Panel Box until a blue line appears.

Figure 16.22 Adding the Panel Box

Note: Make sure that you drag the Panel Box to the H List Box shown in Figure 16.22.

---

2. In the Properties panel, change the Title to Cast Selected Columns into Roles and press Enter.

3. Drag an H List Box (under Containers) over the Panel Box arrow.

Create the Contents of the Cast Selected Columns into Roles Box

1. Drag a Button Box (under Buttons) over the H List Box arrow.

2. In the Properties panel:

    – Change the Title to "Y" and press Enter.

    – In the Press box, type "SetY" and press Enter.

    You copy and paste the script for this button later.

3. Drag a Col List Box (under Input) inside the right border of the Y Button Box.

Figure 16.23 Adding the Col List Box

4. In the Properties panel:

    –   Change the Variable Name to "YList" and press Enter.

        The script that you add later refers to this variable name.

    –   Change the Data type to Numeric.

        The words "optional numeric" appear in the Col List Box.

Tip: Change Min Items to display the minimum number of required items in the Col List Box. If you type 3, then "required numeric" appears three times in the box as a hint to the user.

Create the Action Panel

1. Drag a Panel Box (under Containers) to the right of the Panel Box that you just added.

Figure 16.24 Adding a Panel Box

2. In the Properties panel, change the Title to "Action" and press Enter.
3. Drag a Button Box (under Buttons) over the Panel Box arrow.
4. In the Properties panel:

    –   Change the Title to OK and press Enter.

    –   In the Press box, type "OnOK" and press Enter.
5. Drag another Button Box (under Buttons) below the OK button.

Figure 16.25 Adding a Cancel Button

6.   In the Properties panel:

   –   Change the Title to "Cancel" and press Enter.

   –   In the Press box, type "OnCancel".

7.   Drag a Spacer Box (under Spacers) below the Cancel button.

8.   Drag a Button Box (under Buttons) below the Spacer Box.

9.   In the Properties panel:

   –   Change the Title to "Remove" and press Enter.

   –   In the Press box, type "OnRemove" and press Enter.

10. Select the outer border of the objects on the workspace.

11. Right-click and select **Move to Corner**.

   The extra space around the selected object is removed.

12. Select **File > Save** to save your changes in the.jmpappsource file.

Create the Launch Module Script

1.   Go back to the Launcher with Report sample application window.

2.   Click the Scripts tab.

3.   Select LaunchModule as the Namespace and copy the script.

4.   Go back to your application window and click the Scripts tab.

5.   Delete the script in the LaunchModule namespace and paste the script from the sample application.

6.   Select **File > Save** to save your changes in the.jmpappsource file.

Create the Report Module

The Report Module contains an object that builds a graph.

1.   In your application window, click the button to the right of the LaunchModule tab to add a new module.

2.   In the Properties panel, change the Variable Name to "ReportModule" and press Enter.

3.   Make sure that the Module Type is Report.

4. Deselect Auto Launch to prevent the report from opening when the application is run.

5. Drag an Outline Box (under Containers) to the workspace.

6. In the Properties panel, change the Title to "Radviz" and press Enter.

7. Drag a Graph Box (under Display) over the Outline Box arrow.

8. In the Properties panel, change the Variable Name to "Graph" and press Enter.

9. Select the outer border of the objects on the workspace.

10. Right-click and select Move to Corner.

    The extra space around the selected object is removed.

11. Select **File > Save** to save your changes in the.jmpappsource file.

Create the Report Module Script

1. Go back to the Launcher with Report sample application window.

2. Make sure that the Scripts tab is displayed.

3. Select ReportModule as the Namespace and copy the script.

4. Go back to your application window and click the Scripts tab.

5. Delete the script in the ReportModule namespace and paste the script from the sample application.

6. Select **File > Save** to save your changes in the.jmpappsource file.

Your application should look like the Launcher with Report sample application.

Create the Radviz Script

1. Go back to the Launcher with Report sample application window.

2. Make sure that the Scripts tab is displayed.

3. Select Radviz as the Namespace and copy the script.

4. Go back to your application window and click the Scripts tab.

5. Delete the script in the Radviz namespace and paste the script from the sample application.

6. Select **File > Save** to save your changes in the.jmpappsource file.

Test and Save the Final Application

1. In your application window, click the Application Builder red triangle and select **Run Application**.

2. Select the Sepal length and width and Petal length and width columns, click **Y**, and then click **OK**.

A graph of the data should appear.

Figure 16.26 Iris.jmp Graph

If the graph does not appear or errors are generated, make sure that you typed in the correct variable names when necessary. Also make sure that you added the scripts to the Scripts tab for each module.

3. On Windows, select **File > Save As**, select **JMP Application File** from the Save as type list, and then click **Save**.

   On macOS, select **File > Export**, select JMP Application, and click **Next** to save the application.

   Your newly created application should match the Launcher with Report sample application.

4. Distribute the .jmpapp file to users. This file cannot be modified.

5. Save the .jmpappsource file so you can edit the application and resave it as a JMP application.

About the Application Scripts

When you create an application, you write scripts to provide functionality to display boxes. You can right-click a display box and select a script from the Scripts menu.

Figure 16.27 Adding a Script Interactively

Selecting a script from the Scripts menu adds a placeholder script to the Scripts tab for the selected module. For example, the Press script shown in Figure 16.27 adds the following script to the Scripts tab:

```
Button5Press=Function({this},
    // This function is called when the button is pressed
    name = this << Get Button Name;
);
```

Fill in the placeholder script with your own JSL. In the application that you previously created, you named two variables `SetY` and `ColumnList`. Rewrite the preceding placeholder script:

```
SetY = Function({},
    // This function will be called when the Y button is
    // pressed
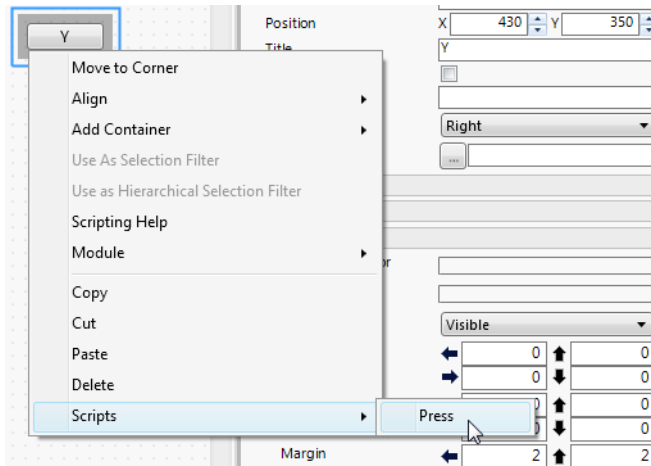
    items = ColumnList << Get Selected;
    YList << Append(items);
);
```

`Get Selected` is passed to the `ColumnList` because you gave the Col List Box this variable name (`ColumnList`) in the object properties. `Get Selected` returns a list of the selected columns. `Append(items)` appends the selected columns to YList, the variable assigned to the Col List Box in the Cast Selected Columns into Roles window.

As shown when you previously created the application, you can write the script from scratch and indicate the script name in the object properties. To get help with syntax for a selected object, right-click the object and select Scripting Index.

Compile Add-Ins with the JMP Add-In Builder

A JMP add-in is a JSL script that you can run anytime from the JMP Add-Ins menu. You can create submenus to group your JMP add-ins and have many levels of menus, if desired.

Experienced JSL script writers can create scripts that extend JMP in many ways (for example, add a custom analytical tool or a user interface to communicate with a database). The JMP add-in architecture simplifies deploying and using these complicated scripts.

You can send co-workers a suite of scripts and tell them how to run them. Another option is sending a single add-in file that any JMP user can install and then use just like any other part of JMP.

As a JMP user, you might be given add-ins to use by co-workers. You can also find add-ins on the JMP website at https://www.jmp.com/addins.

Compile Scripts into an Add-In Using Add-In Builder

To create a JMP add-in, you write the scripts and gather files that are used in the add-in. Then you compile the scripts into an add-in.

- On Windows, select **File > New > Add-In**.
- On macOS, select **File > New > New Add-In**.
- In Dashboard Builder, click the red triangle and select **Save Script > To Add-In**.

The process of compiling scripts into an add-in involves the following steps:

- "Add General Information" on page 889
- "Create Menu Items" on page 891
- "Create Toolbars (Windows Only)" on page 892
- "Specify Start-up and Exit Scripts (Optional)" on page 893
- "Add Additional Files" on page 893
- "Save the Add-In" on page 893
- "Test the Add-In" on page 893

Add General Information

First, in the **General Info** tab, add the general information that identifies and sets up your add-in.

Figure 16.28 Add-In Builder General Info Tab

1. Enter a name for the add-in.

    This is the name of the registered add-in, which appears in the **View > Add-Ins** window.

2. Enter a unique identifier string.

    Unique ID strings are case-insensitive. To ensure uniqueness, it is strongly recommended to use reverse-DNS names (for example, `com.mycompany.myaddin`). The ID string must meet the following requirements:

    – It can be up to 64 characters in length.

    – It must begin with a letter.

    – It should consist only of letters, numbers, periods, and underscores.

    – It should contain no spaces.

    In JSL, use this string to refer to the add-in (for example, in calls to `Get Addin()` or as an argument to $ADDIN_HOME).

3. Enter the version of the add-in.

    If you decide to make changes to the add-in at a later date, you can update the version number and then verify that users have the correct version.

4. Select the minimum version of JMP that the add-in works on.

    ___

    Note: Add-ins were introduced in JMP 9, so no prior versions are supported. And when saving an application as an add-in, select 10 or 11 as the JMP minimum version.

    ___

5. Select whether you want the add-in to be supported on Windows, macOS, or both.

6. (Optional) Select the check box next to **Install after save** if you want to install the add-in after saving it.

    If you do not select this option, the add-in is not installed once you save it, and it does not appear as a selectable menu item in the **Add-Ins** menu.

Create Menu Items

1. Click the **Menu Items** tab.

Figure 16.29 Add-In Builder Menu Items Tab

2. (Optional) Click **Add Submenu**.

   If you have multiple add-ins, you can group them under a submenu.

3. If you add a submenu, next to **Menu item name**, enter the name of the submenu.

   This name appears in the **Add-Ins** menu.

4. Click **Add Command**.

5. Next to **Menu item name**, enter the name of the add-in command.

6. (Optional) Next to **Tooltip for menu item**, enter the content that appears as a tooltip when the users place their cursor over the menu item.

7. Add the script. Select either **Run this JSL** and copy and paste in your script, or select **Run JSL in this file** and click **Browse** to find the file containing your script.

8. (Optional) Select **Use the "Here" namespace for unqualified JSL variable names** to ensure that all unqualified JSL variables are in the Here namespace, and local only to the script.

Notes:

– If your script creates a custom menu or toolbar, the variables are in the Here namespace by default.

– For more information about the Here namespace, see "Advanced Scoping and Namespaces" on page 272 in the "Programming Methods" chapter.

9. (Optional) Browse to add an icon that appears next to the menu item in the **Add-Ins** menu.

10. (Optional, Windows only) Create a keyboard shortcut for the add-in.

11. To add multiple menu items, repeat the steps.

   You can add multiple levels of submenus and add-in commands.

12. Click **Save** and save the add-in to the desired directory.

13. Click **Close**.

Create Toolbars (Windows Only)

1. Click the **Toolbar** tab.

Figure 16.30 Add-In Builder Toolbar Tab

2. Click **Add Command**.

3. Next to Caption, type the name of the toolbar item. This name is only for internal use. The user does not see it.

4. Add the script. Select either **Run this JSL** and copy and paste in your script, or select **Run JSL in this file** and click **Browse** to find the file containing your script.

5. (Optional) Select **Use the "Here" namespace for unqualified JSL variable names** to ensure that all unqualified JSL variables are in the Here namespace, and local only to the script.

6. Click **Save** and save the add-in to the desired directory.

7. Click **Close**.

Note: For more information about customizing menus in JMP, see *Using JMP*.

Specify Start-up and Exit Scripts (Optional)

Click the **Start-Up Script** tab to add a script that runs when JMP starts up (and the add-in starts). You can select an existing script (**Run JSL in this file**) or copy and paste in a script (**Run this JSL**). For example, you could provide a message telling the user that the add-in is installed upon start-up.

Click the **Exit Script** tab to add a script that runs when JMP exits or when you disable the add-in. You can select an existing script (**Run JSL in this file**) or copy and paste in a script (**Run this JSL**). For example, you could provide a prompt for the user to export a JMP data table upon exiting or disabling the add-in.

Add Additional Files

If your script calls other scripts, or contains graphics or data tables, add those files here.

Save the Add-In

Save the add-in by clicking the **Save** button on any tab. This effectively creates the add-in.

• If you selected the **Install after save** option in the General Info tab, then the add-in menu item appears in the **Add-Ins** menu immediately.

• If you did not select the **Install after save** option, when you open the saved add-in file, you are prompted to install the add-in.

Test the Add-In

Once your add-in is installed, test your add-in:

1. Select **View > Add-Ins**.

2. Select your add-in and click **Unregister**.

3. Reinstall your add-in by either selecting **File > Open** in JMP, or by double-clicking your .jmpaddin file.

4. Ensure that the menu and toolbar button run your script correctly and that the script itself runs correctly.

Edit an Add-In

To edit a saved add-in:

1. Select **File > Open**.
2. Navigate to and select the add-in file.
3. Select one of the following options:
   – On Windows click the arrow to the right of the **Open** button and select **Open using Add-In Builder**.
   – On macOS, select the **Edit after opening** option.
4. Click **Open**.

   The file opens in the Add-In Builder. Update the arguments and then save the changes.

Share an Add-In

Once you have a .jmpaddin file, you can share the file with other users. Email the file or you place it in a shared location, such as a network folder, or on the JMP File Exchange (located online in the JMP User Community).

When JMP users open the file, the add-in files are extracted into the appropriate folder, and the add-in is registered and installed. The add-in now appears in the user's JMP Add-Ins menu.

Installing Multiple Add-Ins

If you want to install multiple add-ins, copy the add-ins into the following location:

• Add-In files on Windows are located here:
   – %ALLUSERSPROFILE%/SAS/JMP/AddIns (any user on this machine can access the add-in)
   – %APPDATA%/SAS/JMP/AddIns (only the current user on this machine can access the add-in)
• Add-In files on macOS are located here:
   – /Library/Application Support/JMP/AddIns (any user on this machine can access the add-in)
   – ~/Library/Application Support/JMP/AddIns (only the current user on this machine can access the add-in)

When JMP starts, the addinRegistry.xml file is read, which contains information about previously registered JMP add-ins. Then JMP looks in the add-in folders for any other add-ins and installs them automatically.

Notes:

• The Home Folder for discovered add-ins does not have to be the AddIns subfolder in which the addin.def file was found. The addin.def file can be the only file in that subfolder and have a Home setting that points to some other location where the add-in files actually reside.

• If an automatically discovered add-in has the same unique ID as an add-in that was explicitly registered, the automatically discovered add-in is used.

Register an Add-In Using JSL

If your add-in files are not contained within a .jmpaddin file, you can use the `Register Addin()` JSL function to manually register the addin.def file. This installs and registers the add-in.

• For information about the JSL functions, see the *JSL Syntax Reference*.

• For information about creating the addin.def file, see "Create an Add-In Manually" on page 895.

Notes:

• JMP might find a file named addin.def in the specified home folder. If so, values from that file are used for any optional arguments that are not included in the `Register Addin()` function.

• The addin.def file is used only for values that are not provided in the `Register Addin()` function. This function is useful while developing, but not necessary, since the addin.def file is enough to register an add-in.

Create an Add-In Manually

The addin.def file is a simple text file containing name-value pairs that provide registration information about a JMP add-in. Here are the name-value pairs to include in the addin.def file:

name   Optional. The name that can be displayed in the JMP user interface wherever add-in names are displayed, instead of the unique ID. This name is displayed if no localized names are provided or when JMP is run under a language for which you did not provide a localized name. Named *Add-In Name* in Add-In Builder.

name_xx   Optional. Allows the user-friendly name to be localized for different languages, where *xx* is the two-letter ISO 639-1 code for the language. If you include localized names,

you should still include a language-neutral name in case JMP is running under regional settings for which you do not have a localized name. Not available in Add-In Builder.

id   Required. The unique ID for your add-in. The string can contain up to 64 characters. The string must begin with a letter and contain only letters, numbers, periods, and underscores. Reverse-DNS names are recommended to increase the likelihood of uniqueness. Named *Add-In ID* in Add-In Builder.

home   Optional. The path to the add-in files. The Home Folder for the add-in is assumed to be the folder where addin.def is located. You need to include a setting for home only if the Home Folder is somewhere else (for example, a network shared folder). Not available in Add-In Builder.

home_win   Optional. The path to the add-in files to be used when JMP is running on Windows. Overrides the value specified for home on Windows, if any. Not available in Add-In Builder.

home_mac   Optional. The path to the add-in files to be used when JMP is running on the macOS. Overrides the value specified for home on macOS, if any. Not available in Add-In Builder.

autoLoad   Optional, Boolean. The default value is True (1). Determines whether this add-in is initially configured to load automatically during JMP start-up. Named *Install after save* in Add-In Builder.

host   Optional. Valid values are Win and Mac. Named *Supported Host* in Add-In Builder.

minJMPVersion   Optional. Valid values are integers corresponding to the JMP major version that is the minimum version that the add-in supports. Named *Minimum JMP Version* in Add-In Builder.

maxJMPVersion   Optional. Valid values are integers corresponding to the JMP major version that is the maximum version that the add-in supports. Use this setting *only* if there is a known incompatibility between your add-in and a specific version of JMP. You should provide a new version of the add-in for later versions of JMP. Not available in Add-In Builder.

Example of a addin.def File

```
id="com.mycompany.myaddin"
name="My Add-In's Friendly Name"
name_fr="My Add-In's French Name"
name_de="My Add-In's German Name"
home="\\server\share\myjmpaddin"
Autoload=1
MinJMPVersion=9
```

Example of a JMP Add-In

A sample add-in named Simple Calculator.jmpaddin is located in one of the following folders:

- On Windows: C:/Program Files/SAS/JMP/16/Samples/Scripts
- On macOS: /Library/Application Support/JMP/16/Sample/Scripts

Note: On Windows, in JMP Pro, the "JMP" folder is named "JMPPro".

To see what the add-in contains, change the extension to .zip and unzip it into a new folder. To see how it works, change the extension back to .jmpaddin and install.

The add-in contains the following files:

addin.def

Provides the specification for JMP to register the add-in. It contains only these two lines:

```
id="com.jmp.sample.calculator"
name="Simple Calculator"
```

addin.jmpcust

Provides the menu customization file that is created when you interactively create a custom menu. Your customizations are shown only to the current user; other users who log on to the computer and open JMP do not see your personal toolbars and menus. This example places the add-in menu item into the default **Add-Ins** menu.

calculator.jsl

A JSL script that creates a basic calculator.

calc_icon.gif

The image used as the calculator's icon.

To download more add-ins, visit
https://community.jmp.com/t5/File-Exchange/ct-p/FileExchange.

Manage JMP Add-Ins

Install Add-Ins

A JMP add-in is a file that has the .jmpaddin extension. You can install it in one of two ways:

1. Select **File > Open**.
2. Navigate to the .jmpaddin file and select it.

3. Click **Open**.

You can also double-click the .jmpaddin file or drag it onto the JMP Home Window.

View Your Add-Ins

Select **View > Add-Ins** to see the add-ins that you have already installed.

Reorder Your Add-Ins

Select **View** > **Add-Ins**, then use the Load Order arrows to move each add-in where you want it to appear in the Add-Ins menu.

Update Add-Ins

If you are given an update to an add-in you already have installed, you can just install the update as you did the original add-in. Doing so overwrites the old version with the new one.

Disable, Enable, and Remove Add-Ins

To temporarily disable an add-in without removing it

1. Select **View > Add-Ins**.
2. Select the add-in that you want to disable.
3. Deselect the **Enabled** check box.

To enable a disabled add-in

1. Select **View > Add-Ins**.
2. Select the add-in that you want to enable.
3. Select the **Enabled** check box.

To unregister an add-in

1. Select **View > Add-Ins**.
2. Select the add-in that you want to remove.
3. Click **Unregister**.

Chapter **17**

Common Tasks
Getting Started with Sample Scripts

Examining working scripts line-by-line is one of the best ways to learn JSL. This chapter describes common tasks in JMP, such as converting date/time values and extracting specific values from reports. Sample scripts that address these issues are installed with JMP in the Samples/Scripts folder so you can run them yourself.

Contents

Run a Script at Start Up

You can run the same script every time you start JMP. For example, you might include the definitions of some utility functions or a namespace that you want available throughout your JMP session. You might also set some preferences in a startup script to keep preferences consistent. Preferences persist across JMP sessions, but you could explicitly reset some preferences in case they were changed at some point in the previous session.

Name the script jmpStart.jsl and place it in one of the following folders, as appropriate for your operating system. When JMP starts, JMP looks for the jmpStart.jsl script in these folders in the order in which they are listed here. The first one that is found is run, and the search immediately stops.

Note: Some path names in this section refer to the "JMP" folder. On Windows, in JMP Pro, the "JMP" folder is named "JMPPro".

On Windows:

1. C:/Users/<username>/AppData/Roaming/SAS/JMP/16
2. C:/Users/<username>/AppData/Roaming/SAS/JMP

On macOS:

1. /Users/<username>/Library/Application Support/JMP/16
2. /Users/<username>/Library/Application Support/JMP

The jmpStart.jsl script runs only for a particular user on a computer. You can add a script named jmpStartAdmin.jsl in one of the following places, as appropriate for your operating system. This script is run for every user on a computer. JMP runs jmpStartAdmin.jsl first if found. Then JMP runs jmpStart.jsl if found.

On Windows:

1. C:/ProgramData/SAS/JMP/16
2. C:/ProgramData/SAS/JMP

On macOS:

1. /Library/Application Support/JMP/16
2. /Library/Application Support/JMP

Convert Character Dates to Numeric Dates

Data might appear to be numeric in the data table. However, the column properties may specify a character data type. To manipulate the data as date/time values, convert the column to a numeric column and specify how you want the values to appear.

Convert Dates.jsl creates a data table, specifies the data input format, changes the column to a numeric continuous column, and applies the m/d/y format (Figure 17.1).

```
// Create a data table with character dates.
dt = New Table( "Example",
    Add Rows( 3 ),
    New Column( "Dates",
        Character,
        Nominal,
        Set Values( {"25/01/2010", "30/09/2009", "15/12/2013"} )
    )
);

// Display a modal dialog for the user to confirm the format conversion.
nw = New Window( "Date Conversion",
    <<Modal,
    tb = Text Box(
        "Notice the d/m/y format of the character dates.
Click OK to convert the column to a numeric column and apply the m/d/y
    format."
    )
);

/* Apply the Numeric data type.
Specify the Informat (input format) value "d/m/y".
Specify the Format (display format) value "m/d/y".
Apply the Continuous modeling type */
col = Column( dt, "Dates" );
col << Data Type( "Numeric", Informat( "d/m/y" ), Format( "m/d/y" ) );
col << Modeling Type( "Continuous" );
```

Figure 17.1 Converting Character Dates (Before and After)

When you change the column's data type from character to numeric, defining the format in which the data were entered is important. In this example, Informat("d/m/y") defines the input format. Format("m/d/y") defines the new display format. If Informat() is omitted, the Format() value is applied as both the input and display format. This results in missing values for some data.

Modify Convert Dates.jsl to see for yourself.

1. Open Convert Dates.jsl from the sample scripts folder.

2. Right-click the script window and select **Show Line Numbers**.

3. On line 9, change "25/01/2010" to "01/25/2010".

4. On line 27, delete Informat("d/m/y"), (including the comma).

5. Run the script.

   Format("m/d/y") is applied to the column. Only "01/25/2010" appears in the column. The other values are missing; "30/09/2009" and "15/12/2013" are not valid m/d/y values.

Format Date-Time Values and Subset Data

How can you work with dates in JMP? JMP provides a number of formats that you can use to make comparisons and then subset data based on the date.

Select Where Using Dates.jsl applies the Date MDY format to a column of departure dates and subsets the data. A summary table of mean net costs by departure date then appears (Figure 17.2).

```
/*  How can you work with dates in JSL? JMP provides a number of formats
that you can use to make comparisons and then subset data based on the date.
*/

hdt = Open( "$SAMPLE_DATA/Travel Costs.jmp" );
```

```
/* Apply the Date MDY format to Departure Date values and then select only
February dates. */
hdt << Select Where(
    (Date MDY( 02, 01, 2007 ) <= :Departure Date < Date MDY( 03, 1, 2007 ))
    );

/* Subset the selected rows into two tables: one table contains February
departure dates, the other contains all data for those departure dates. */
nt1 = hdt << Subset( Columns( :Departure Date ),
   Output Table Name( "February Departure Date" ) );
nt2 = hdt << Subset( Output Table Name( "February Data" ) );

/* Create a summary table, grouping mean cost by day of week that departure
took place. */
sumDt = nt2 << Summary(
   Group( :Departure Day of Week ),
   Mean( :Net Cost ),
   Output Table Name( "Mean Net Cost by Departure Date" )
);
```

Figure 17.2 The Original Table and the Final Summary Table

Create a Formula Column

How do you create a formula column that combines conditional expressions with value comparisons? Create a Formula Column.jsl shows how to create a new formula column that evaluates ages in Big Class.jmp and returns the result in the new column (Figure 17.3).

```
/* Scenario:

How do you create a formula column that combines conditional expressions
with value comparisons? This script shows how to create a new formula
column that evaluates ages in Big Class.jmp and returns the conditional
result in the new column. */

dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

/* Create a new character column for the formula.
Insert "early" in the new column if the age is less than or
equal to 12. Insert "mid" if the age is less than or equal to
15 but greater than 12. For ages greater than 15, insert "late".
*/

dt << New Column( "Adolescent Phase",
  Character,
  Formula(
    If( :age <= 12, "early",
      12 < :age <= 15, "mid",
      "late"
      )
    )
);
```

Figure 17.3 Conditional Expression in Formula

Extract Values from an Analysis into a Report

How do you capture specific results of an analysis into a custom report using JSL?

The JMP platforms in the Analyze and Graph menus contain two objects known as the analysis and report layers. Messages are sent to the analysis layer that generate the desired results.

Extract Values from Reports.jsl performs a Bivariate analysis and shows results such as the sample size, RSquare, and Correlation in a new report window. Figure 17.4 shows the Bivariate report along with the customized report, though the Bivariate report window closes after the script is finished.

```
/* Scenario: How do you capture specific results of an analysis
in a report using JSL?
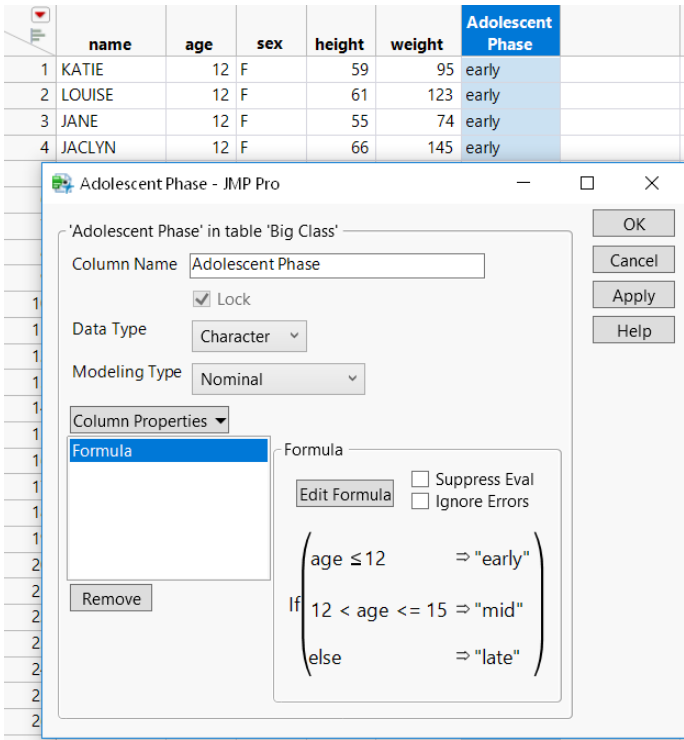
The JMP platforms in the Analyze and Graph menus contain two objects
known as the analysis and report layers.

Messages are sent to the analysis layer that generate the desired results.

This script performs a Bivariate analysis and shows results
such as the sample size, RSquare, and Correlation in a new report window.
*/

sd = Open( "$SAMPLE_DATA/Lipid Data.jmp" );

biv = Bivariate(           // biv is the analysis layer.
    Y( :Triglycerides ),
    X( :LDL ),
    Density Ellipse( 0.95, {Line Color( {213, 72, 87} )} ),
    Fit Line( {Line Color( {57, 177, 67} )} ),
);

/* Make sure the second Outline Box (called "Correlation")
in the Bivariate report is open. You can then see which content
is extracted into the Custom report. */
report(biv) [Outline Box( 2 )] << Close( 0 );

reportbiv = biv << Report;  // reportbiv is the report layer.

/* The density ellipse is generated first.
Extract the correlation coefficient. */
corrvalue = reportbiv[Outline Box( 2 )][Number Col Box( 3 )] << Get( 1 );

/* ...followed by Fit Line
Extract the numeric values from the Summary of Fit report
and place them in a matrix. */
sumfit = reportbiv[Outline Box( 4 )][Number Col Box( 1 )] << Get as Matrix;

// Extract the values of RSquare and AdjRSquare as one by one matrices.
rsquare = sumfit[1];
adjrsq = sumfit[2];
avg = sumfit[4];
samplesize = sumfit[5];

/* Extract the first column of the Parameter.
Estimates report as two objects. */
term = reportbiv[Outline Box( 7 )][String Col Box( 1 )] << Get();
```

```
// Clone the report layer as a String Col Box.
cloneterm = reportbiv[Outline Box( 7 )][String Col Box( 1 )] << Clone Box;

// Extract the Parameter Estimates values as a matrix.
est = reportbiv[Outline Box( 7 )][Number Col Box( 1 )] << Get as Matrix;

// Extract the Standard Error values as a matrix.
stde = reportbiv[Outline Box( 7 )][Number Col Box( 2 )] << Get as Matrix;

dvalues = [];
dvalues = samplesize |/ adjrsq |/ rsquare |/ corrvalue;
sfactor = term[2];

dlg = New Window( "Custom Report",
  Outline Box( "Selected Values",
     /* The Lineup box defines a two-column layout, each of which contains
     a Text Box. */
     Lineup Box( N Col( 2 ),
     Text Box( "Factor of Interest: " ),
     Text Box( sfactor ), ),
     tb = Table Box(
        /* Display an empty string in the first column
           and the text in the second column. */
        String Col Box( "  ",
           {"Sample Size: ", "Adjusted RSquare: ", "RSquare: ",
           "Correlation:"}
        ),
        // Insert a 30 pixel x 30 pixel spacer between the columns.
        Spacer Box( Size( 30, 30 ) ),

        ,
        /* Display an empty string in the first column
        and the dvalues in the second column. */
        Number Col Box( "  ", dvalues )
     ),
     // Insert a 1 x 30 spacer.
     Spacer Box( Size( 0, 30 ) ),

     ,
     Table Box(
        /* Display the cloned String Col Box followed by a spacer.
           Then insert the Parameter Estimates and Standard Error values. */
        CloneTerm,
        Spacer Box( Size( 10, 0 ) ),

        ,
        Number Col Box( "Estimate", est ),
        Spacer Box( Size( 10, 0 ) ),
```

```
                    ,
            Number Col Box( "Standard Error", stde )
        )
    )
);
tb << Set Shade Headings( 0 ); // Turn off shaded table headings.
tb << Set Heading Column Borders( 0 ); // Turn off table column borders.
Close( sd ); // Close the data table.
```

Figure 17.4 Customized Report from the Bivariate Analysis

Create an Interactive Program

How do you gather numeric input from the users, perform a calculation on that input, and show the results in a new window?

Prime Numbers.jsl asks the user to enter a number and then factors the number or confirms it as a prime number (Figure 17.5). This script is a good example of aligning several types of display boxes, concatenating text, and working with conditional functions.

```jsl
/* How do you gather numeric input from the users, perform a
calculation on that input, and show the results in a new window?

This script demonstrates how to create an interactive program that
asks the user to enter a number and then factors the number or confirms
it as a prime number. */

// Ask the user to enter a name and number.
nw = New Window( "Factoring Fun",

V List Box(
  Text Box( "Choose a number between 2 and 100, inclusive. " ),
  Spacer Box( Size( 25, 25 ) )
    ),
    V List Box(
        Lineup Box(
            2,
            Text Box( "Your name    " ),
            uname = Text Edit Box( "< name > ", << Justify Text( Center ) ),
            Text Box( "Your choice " ),
            uprime = Number Edit Box( 2 )
        ),
        Spacer Box( Size( 25, 25 ) ),
        H List Box(
            Button Box( "OK",
         // Unload responses.
                username = uname << Get Text;
                fromUser0 = uprime << Get;

        // Test input for out of range condition.
                If( fromUser0 <= 1 | fromUser0 > 100,
          // Send message to user that input value is out of range.
                nw2 = New Window( " Factoring Fun: Message for "
                    || username,
                      <<Modal,
                      Text Box(
                          "The number you chose, " || Char( fromUser0 ) ||
```

```
                    " is not between 2 and 100, inclusive.  Please try
                    again.  "
             ),
             Button Box( "OK" )

    ),
       /* Else the number is within range.
       Test for a prime number. If not prime, factor it.
       Create a vector which holds the prime numbers
       within specified range. */
     primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
    41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97];
     // Count the number of primes in the vector.
     p# = N Row( primes );

     isprime = 0; // Set flag.
    // Make a copy of the value for processing.
    fromuser1 = fromuser0;
     factors = {}; // Initialize list.

      /* Process the value by checking for prime then
     factoring if needed. */
     While( isPrime == 0,

        // Compare value to vector of prime numbers.
        If( Any( fromuser0 == primes ),
      // If found, place value in factor list.
            Insert Into( factors, fromUser0 );
            isPrime = 1 // Set condition to exit While loop.
            ;
        ); // End For loop.

        If( isprime == 0,
            For( q = 1, q <= p#, q++,
                If( Mod( fromuser0, primes[q] ) == 0,
                    fromUser0 = fromUser0 / primes[q];
                    Insert Into( factors, primes[q] );
                    q = p# + 1 // End if-then loop.
                    ;
                ); // End If loop.
            ); // End For loop.
        );  // End If/Then loop.

    ); // End while loop.

    cfUser0 = Char( fromUser1 );
```

```
                            nf = N Items( factors );
                            If( nf >= 2,
                                fmsg = "The number you have chosen has the following
                                 factors: ",
                               fmsg = "The number you have chosen is a prime number: "
                            );
                // Show message to user about results.
                            nw3 = New Window( " Factoring Fun - Your Results",
                                <<Modal,
                                Text Box( username || ", you chose: " || cfUser0 ),
                                Spacer Box( Size( 25, 25 ) ),
                                Text Box( fmsg || " " || Char( factors ) ),
                                Spacer Box( Size( 25, 25 ) ),
                                Button Box( "OK" )
                            );
                        );
                    ), // End the main OK button script.
                    // Close the window and the program.
                    Button Box( "Cancel", nw << Close Window )
                )
            )
        );
```

Figure 17.5 Factor Numbers Interactively
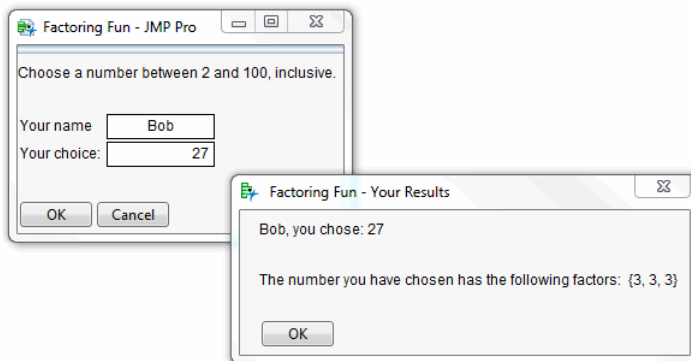
Compatibility Notes

This appendix provides information about changes that affect compatibility in JSL scripts. Please update your scripts accordingly so that the scripts run correctly in the latest version of JMP.

Compatibility Issues in 16.0

Copy File(), Move File(), Rename File(), and Delete File()

In JMP 15.0, copying a file, moving a file, renaming a file, and deleting a file that didn't exist threw an error. Now, it returns 0.

Row Selection() and Case Sensitivity

`Match Case`, `Match Case()`, `Match Case(1)` and a missing `Match Case` do a case sensitive match in `Row Selection()`. `Match Case(0)` does a case insensitive match. Previously, `Row Selection()` always matched case sensitively even if `Match Case` was not supplied.

Target Level Column Property

You can use the new Target Level column property to specify the level of interest in binary logistic regression models. The default level is High. The main effects will be in the slope of the logistic probability curve, the signs of the linear model estimates, the profiler shape, and the ROC target. Be aware that some scripts that assume the target level in Fit Model or Fit Y by X logistic regression models should be checked.

Opening Multiple Data Tables

`Open()` returns a list of data tables if multiple data tables are created (for example, when importing PDF, HDF5, or Excel files). Previously, the `Open()` command returned a single table, even if multiple tables were created in the process.

Format(), Informat(), and Variations

The results of `Format()`, `Informat()` and their variations are written to the log if the format is invalid. Previously, they returned `m/d/y` when the format was invalid.

Graph Background Color Preference in Contour Plots

The Graph Background Color preference no longer affects the Contour Plot legend background color. Previously, the background color was applied to both the graph background and the legend background.

Appendix **B**

Efficient Scripts

Tips for Writing Scripts That Run Faster

This appendix provides ideas for writing scripts that run faster and more efficiently. For example, it is possible to control how much time is allocated to finding an optimal design. You can collapse outlines in a report that are not necessary. Or you can write static matrices, which is faster than developing them dynamically.

Contents

Using Matrices

- The matrix concatenation operators, || and |/, add columns or rows to a matrix dynamically. However, the script runs faster if you define a matrix in a static fashion instead.

```
// faster example
t1 = Tick Seconds();
A = J( 10, 10000, . );
For( i = 1, i <= 10000, i++,
   a2 = J( 10, 1, Random Gamma( 1, 1 ) );
   // define a2 to be a 10 by 1 vector of random gamma values.
        A[0, i] = a2;
        /* populate the ith column of the matrix A to have the
           values in a2.  The first argument, 0, means "all rows". */
);
t2 = Tick Seconds();
Show( t2 - t1 );
   t2 - t1 = 0.0166666666627862;

// slower example
t3 = Tick Seconds();
B = J( 10, 0, . );
For( i = 1, i <= 10000, i++,
   a2 = J( 10, 1, Random Gamma( 1, 1 ) );
   B = B || a2;
);
t4 = Tick Seconds();
Show( t4 - t3 );
   t4 - t3 = 3.1333333333605;
```

- If you know the size of a matrix that you're going to need, it's typically better to allocate the matrix in its entirety first and then populate it. In the example below, the J() function is doing the allocation of the 100000x1 matrix all at the same time.

```
Delete Symbols( a1, a2 );
// faster example
t1 = Tick Seconds();
a1 = J( 100000, 1, Random Uniform() );
t2 = Tick Seconds();
Show( t2 - t1 );
Delete Symbols( a1 );
   t2 - t1 = 0;

// slower example
t3 = Tick Seconds();
a2 = J( 0, 0 );
```

```
For( i = 1, i <= 100000, i++,
    a2 |/= Random Uniform()
);
t4 = Tick Seconds();
Show( t4 - t3 );
    t4 - t3 = 5.05000000004657;
```

- Use matrices rather than a data table to store numbers.

Design of Experiments

In DOE, there are options such as Number of Starts and Design Search Time that enable
you to control how much time is allocated to finding an optimal design.

```
// Number of Starts example
DOE(
    Custom Design,
    Add Factor( Continuous, -1, 1, "X1", 0 ),
    Add Factor( Continuous, -1, 1, "X2", 0 ),
    Add Factor( Continuous, -1, 1, "X3", 0 ),
    Number of Starts( 1000 ),
    Make Design
);
// Design Search Time example
d = DOE( Screening Design );
dt2 = Open( "$SAMPLE_DATA/Design Experiment/Weld Factors.jmp" );
d << Load Factors;
Close( dt2 );
d << {Set Random Seed( 12345 ), Screening Type( 1 ),
Suppress Cotter Designs, Design Search Time( 8 ),
Number of Column Starts( 50 ), Set Sample Size( 12 ), Make Design};
```

Scripting Data Tables

- Displaying a large data table can cause scripts to run slower. To speed things up, you can
  make a data table invisible, especially if you need to reference numbers in a report that
  will be used for future calculations or used in another platform. In this case, there isn't a
  need to see the current data table.

  ```
  dt = Open( "$SAMPLE_DATA/Big Class.jmp", invisible);
  ```

- Drawing objects is slow. Suppose that you have a graph with a lot of data. The platform
  does complicated calculations or the data table contains a formula. Adding a row can
  require a lot of processing. Use Begin Data Update() so you only redraw the graph once

instead of for each change. `Begin Data Update()` saves all update messages until `End Data Update()` is reached.

```
dt = Open( "MyData.jmp" );
dt << Distribution(
   Column( :"N=1"n, :"N=5"n), :"N=10"n )
);
Wait(1);
dt << Begin Data Update;
dt << Add Rows( 2000 );
dt << End Data Update;
```

- The use of functions might help speed things up, for example, using the `Summarize()` function rather than creating a Summary table. Maintaining a data table can be too much overhead. See "Store Summary Statistics in Global Variables" on page 364 in the "Data Tables" chapter.

- Data tables are great for keeping tabular data.

  – Keep data table strings under 23 characters. Longer strings have additional overhead.

  – Numeric values use less memory than character values.

Scripting Graphs and Display Boxes

- When displaying large numbers in graphs, you can set a preference to use Fast Markers.

  ```
  Preferences( Fast Marker Threshold( 25000 ) ); // 50000 is the default.
  ```

- Collapse some of the outline nodes in a report to speed up the script.

  ```
  SendToReport(
       Dispatch(
           {"Fit Life by X - Hours BY Arrhenius Celsius (Temp) Regression "},
           "Scatterplot",
           OutlineBox,
           {Close( 1 )}
       ),
  ```

- Construct your display boxes outside `New Window()` and then include them by reference in `New Window()`. This method saves the drawing until the end.

  ```
  tb = Text Box( "Select:" );
  lb = List Box( {"One", "Two", "Three"} );
  New Window( "test", tb, lb );
  ```

- The user might not need to see a report if the numbers will be used in another platform or for a future calculation. You can hide the report.

  ```
  dt= Open( "$SAMPLE_DATA/Big Class.jmp" );
  obj = Bivariate( Y( :Weight ), X( :Height ) );
  ```

```
obj << Show Window( 0 ); // hide the window
obj << Fit Line( {Confid Curves Fit( 1 )} );
```

Submit the following script to see the report window. Notice that the confidence curves were fit even though the window was not shown.

```
obj << Show Window( 1 );
```

Scripting Numbers, Strings, Arrays, and Lists

- Use a faster algorithm. Replacing an O(n^2) algorithm with an O(n) algorithm is faster than anything else. See https://en.wikipedia.org/wiki/Big_O_notation.

- Avoid concatenating long strings. 100s of characters is OK; 1,000,000s of characters will be slow if you build the string in 1,000,000s of operations.

- Use numeric arrays (matrices, not lists).
  - Use numeric arrays with operators that operate on the entire array.
  - Avoid writing loops to manipulate numeric arrays.

- Use associative arrays for lookups.

Programming Tips

- Avoid `Break()`, `Continue()`, and `Return()` functions for speed. Using them might make it easier for you to read the script.

- Functions versus expressions:
  - User-defined functions have overhead for calling them but might make it easier for you to read the script.
  - Expressions have no overhead for calling but might make it more difficult for you to read the script.

Troubleshooting

Use the JSL Debugger to discover where the script is expending the most time. See "Debug or Profile Scripts" on page 77 in the "Scripting Tools" chapter.

Appendix **C**

References

Morgan, J. (2010). "Expression Handling Functions: Part I." *JMPer Cable* 26:15–19. Accessed
January 2, 2018.
https://www.jmp.com/content/dam/jmp/documents/en/newsletters/jmper-cable/26_winter
_2010.pdf.

Appendix **D**

Glossary

Terms, Concepts, and Placeholders

| In syntax summaries, | means "or" and separates possible choices. Usually choices separated by | are mutually exclusive. In other words, you have to pick one and cannot list several.

argument   An argument is something specified inside the parentheses of a JSL function, message, and so forth. Big Class.jmp is the argument in Open("Big Class.jmp").

You can often infer the meaning by the argument's position. For example, the values *200* and *100* in size(200, 100) are implicit arguments. The first value is always interpreted as the width; the second value is always interpreted as the height. See also *named argument*.

Boolean   A Boolean value is a yes/no value, something that is on or off, shown or hidden, true or false, 1 or 0, yes or no. A value listed as *being a Boolean value* is one that evaluates to true or false (or missing).

col   In syntax summaries, a placeholder for any reference to a data table column. For example, Column("age").

command   A generic description for a JSL statement that performs an action. The *Scripting Guide* prefers the more specific terms *function* and *message* when they are applicable.

current data table   The current data table is the data table that Current Data Table() either returns or is assigned.

current row   The current row for scripting is defined to be zero (no row) by default. You can set a current row with Row() or For Each Row, and so forth.

database   Although the term is much more general, for JMP's purposes, the word "database" describes any external data source (such as SQL) accessed through ODBC with JSL's Open Database command.

Datafeed   A Datafeed is a method to read real-time data continuously, such as from a laboratory measurement device connected to a serial port.

db   In syntax summaries, a placeholder for any reference to a display box. For example, report(Bivariate[1]).

dt   In syntax summaries, a placeholder for any reference to a data table. For example, Current Data Table() or Data Table("Big Class.jmp").

eliding operator   An eliding operator is one that causes arguments on either side to combine and evaluate differently than if the statement were evaluated strictly left to right. For example, 12<a<13 is a range check to test whether *a* is between 12 and 13: JMP reads the whole expression before evaluating. If < did not elide, the expression would be evaluated left to right as (12<a)<13. In other words, it would check whether the result of the comparison (1 or 0, for false or true) is below 13, which of course would always yield 1 for true. The << operator (for object<<message, which is equivalent to Send(object, message) is another example of an eliding operator.

function   A function takes an argument or series of arguments inside parentheses after the function name. For example, the infix operator + has a function equivalent Add(). The statements $3 + 4$ and Add(3, 4) are equivalent. All JSL's operators have function equivalents, but not all functions have operator equivalents. For example, Sqrt(a) can be represented only by the function. Also see the Function operator for storing a function under a name.

global variable   A global variable is a name to hold values that exists for the remainder of a session. Globals can contain many types of values, including numbers, strings, lists, or references to objects. They are called globals because they can be referred to almost anywhere, not just in some specific context.

head   The head of a JSL expression is the expression without any of its arguments. For example, the head of the expression Assign(x, 100.1) is Assign().

infix operator   An infix operator takes one argument on each side, such as + in arithmetic, $3 + 4$, or the = in an assignment, a=7.

L-value   Something that can be the destination of an assignment. In this manual, L-value describes an expression that normally returns its current value but that can alternatively receive an assignment to set its value. For example, you would ordinarily use a function such as Row() to get the current row number and assign it to something else. For example, x=Row(). However, since Row is an L-value, you can also place it on the left side of an assignment to set its value. For example, Row()=10.

list   A list is a multiple-item data type entered in special brace { } notation or with the List function. Lists enable scripts to work with many things at once, often in the place of a single thing.

matrix   A matrix is a JMP data type for a rectangular array of rows and columns of number. In JSL, matrices are entered in bracket [ ] notation or with the Matrix function.

message   A message is a JSL statement that is directed to an *object*, which knows how to execute the message.

metadata   In JMP data tables, metadata are data about the data, such as the source of the data, comments about each variable, scripts for working with the data, and so on.

mousedown   An event generated by pressing down the mouse button. See "Handle()" on page 669 and "Mousetrap()" on page 674.

mouseup   An event generated by releasing the mouse button. See "Handle()" on page 669 and "Mousetrap()" on page 674.

name   A name is a reference to a JSL object. For example, when you assign the numeric value 3 to a global variable in the statement `a=3`, "a" is a name.

namespace   A namespace is a collection of unique names and corresponding values. Namespaces are useful for avoiding name collisions between different scripts.

named argument   A named argument is an optional argument that you select from a predetermined set and explicitly define. For example, `HexToNumber(`*`hextext`*`,` `<Base(`*`number`*`)>)` has a single named argument, `Base`. This is the fixed name that must directly be in the `HexToNumber` argument.

ODBC database   The Microsoft standard for Open DataBase Connectivity. JSL supports access to any ODBC-enabled data source through the `Open Database` command.

obj   In syntax summaries, a placeholder for any reference to an analysis platform. For example, `Bivariate[1]`.

object   An object is a dynamic entity in JMP, such as a data table, a data column, a platform results window, a graph, and so forth. Most objects can receive messages telling them to act on themselves in some way.

operator   Usually operator refers to a one- or two-character symbol such as + for addition or <= for less than or equal to.

POSIX   POSIX is an acronym for Portable Operating System Interface and is a registered trademark of the IEEE. POSIX pathnames enable you to use one syntax for paths for any operating system, instead of having to use a different syntax for each.

postfix operator   A postfix operator takes on argument on its left side (before the operator), such as `a++` for postincrement or `a−−` for postdecrement.

pre-evaluated statistics   Statistics that are calculated once and used as constants thereafter.

prefix operator   A prefix operator takes one argument on its right side (after the operator), such as `!a` for negation.

reference   A way to address a scriptable object in order to send it messages. For example,. `column("age")` or `Current Data Table()` or `Bivariate[1]`. Typically a reference is stored in a *global variable* for convenience.

row state   A data element type to store any combination of the following attributes for data rows: excluded, hidden, labeled, selected, color, marker, hue, shade.

scalar   A simple non-matrix numeric value.

scoping operator   A scoping operator forces a name to be interpreted as a particular type of data element, for example the `:` operator in `:name` forces *name* to be resolved as a column; the `::` operator in `::name` forces *name* to be resolved as a global variable. In the rare case that you do want to share specific variables among all opened projects, use the `:::` root operator.

toggle   Omitting the Boolean argument for a row state command toggles the setting. If the option is off, the message turns it on. If the option is on, the message turns it off. Sending such a command repeatedly flips back and forth between on and off. If you include the Boolean argument, the command sets an absolute on or off state, and sending the command repeatedly has no further effect. For all other messages, omitting the Boolean argument enables the option.

vector   A matrix with only one column or row.

Appendix **E**

Technology License Notices

- Scintilla is Copyright © 1998–2017 by Neil Hodgson <neilh@scintilla.org>.

  All Rights Reserved.

  Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

  NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

- Progressë® Telerik® UI for WPF: Copyright © 2008-2019 Progress Software Corporation. All rights reserved. Usage of the included Progress® Telerik® UI for WPF outside of JMP is not permitted.

- ZLIB Compression Library is Copyright © 1995-2005, Jean-Loup Gailly and Mark Adler.

- Made with Natural Earth. Free vector and raster map data @ naturalearthdata.com.

- Packages is Copyright © 2009–2010, Stéphane Sudre (s.sudre.free.fr). All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  Neither the name of the WhiteBox nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

- iODBC software ia Copyright © 1995–2006, OpenLink Software Inc and Ke Jin (www.iodbc.org). All rights reserved.

- This program, "bzip2", the associated library "libbzip2", and all documentation, are Copyright © 1996–2019 Julian R Seward. All rights reserved.

4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Julian Seward, jseward@acm.org

bzip2/libbzip2 version 1.0.8 of 13 July 2019

- R software is Copyright © 1999–2012, R Foundation for Statistical Computing.

- MATLAB software is Copyright © 1984-2012, The MathWorks, Inc. Protected by U.S. and international patents. See www.mathworks.com/patents. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

- libopc is Copyright © 2011, Florian Reuter. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

– Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

– Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and / or other materials provided with the distribution.

– Neither the name of Florian Reuter nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF

- libxml2 - Except where otherwise noted in the source code (e.g. the files hash.c, list.c and the trio files, which are covered by a similar license but with different Copyright notices) all the files are:

- Regarding the decompression algorithm used for UNIX files:

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

- Snowball is Copyright © 2001, Dr Martin Porter, Copyright © 2002, Richard Boulton.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS \"AS IS\" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Pako is Copyright © 2014–2017 by Vitaly Puzrin and Andrei Tuputcyn.

  Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

  The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

- dmlc/xgboost is Copyright © 2019 SAS Institute.

  Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

  http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

- libzip is Copyright © 1999–2019 Dieter Baron and Thomas Klausner.

  This file is part of libzip, a library to manipulate ZIP archives. The authors can be contacted at <libzip@nih.at>.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  3. The names of the authors may not be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- OpenNLP 1.5.3, the pre-trained model (version 1.5 of en-parser-chunking.bin), and dmlc/xgboost Version .90 are licensed under the Apache License 2.0 are Copyright © January 2004 by Apache.org.

  You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- You must give any other recipients of the Work or Derivative Works a copy of this License; and

- You must cause any modified files to carry prominent notices stating that You changed the files; and

- You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

- If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

- You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

- LLVM is Copyright © 2003–2019 by the University of Illinois at Urbana-Champaign.

  Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

  http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

- clang is Copyright © 2007–2019 by the University of Illinois at Urbana-Champaign.

  Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

  http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS", WITHOUT WARRANTIES OR CONDITIONS OF

ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

- lld is Copyright © 2011–2019 by the University of Illinois at Urbana-Champaign.

  Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

  http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS", WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

- libcurl is Copyright © 1996–2020, Daniel Stenberg, daniel@haxx.se, and many contributors, see the THANKS file. All rights reserved.

  Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

  Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.