Version 12

Scripting Guide

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

Marcel Proust

JMP, A Business Unit of SAS
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2015. *JMP® 12 Scripting Guide*. Cary, NC: SAS Institute Inc.

JMP® 12 Scripting Guide

ISBN 978-1-62959-482-8 (Hardcopy)
ISBN 978-1-62959-484-2 (EPUB)
ISBN 978-1-62959-485-9 (MOBI)
ISBN 978-1-62959-483-5 (PDF)

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

March 2015

Other brand and product names are trademarks of their respective companies.

Technology License Notices

- Scintilla - Copyright © 1998-2014 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.

- Telerik RadControls: Copyright © 2002-2012, Telerik. Usage of the included Telerik RadControls outside of JMP is not permitted.
- ZLIB Compression Library - Copyright © 1995-2005, Jean-Loup Gailly and Mark Adler.
- Made with Natural Earth. Free vector and raster map data @ naturalearthdata.com.
- Packages - 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

(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.

- iODBC software - Copyright © 1995-2006, OpenLink Software Inc and Ke Jin (www.iodbc.org). 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 OpenLink Software Inc. 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 OPENLINK 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.

- bzip2, the associated library "libbzip2", and all documentation, are Copyright © 1996-2010, Julian R Seward. 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.

The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

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.

- 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 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.

- 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 licence but with different Copyright notices) all the files are:

  Copyright © 1998 - 2003 Daniel Veillard. All Rights Reserved.

  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.

  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.IN NO EVENT SHALL THE DANIEL VEILLARD 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 Daniel Veillard shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from him.

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
- a blog with tips, tricks, and stories from JMP staff
- a forum to discuss JMP with other users

http://www.jmp.com/getstarted/

Contents

Scripting Guide

Chapter 1

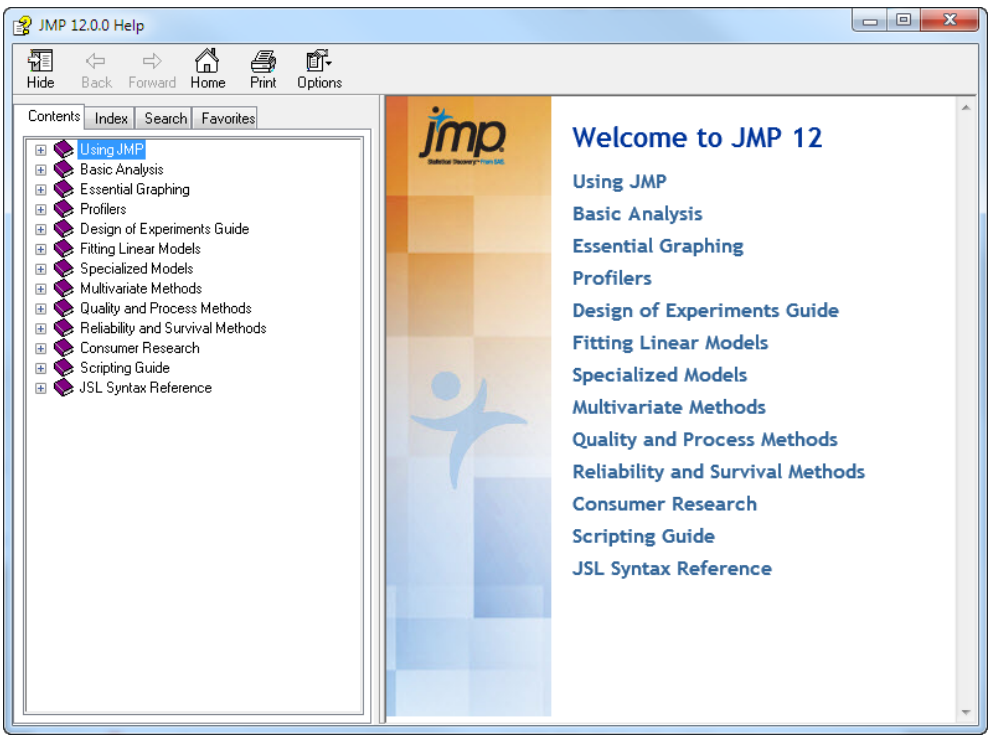
Learn about JMP

Documentation and Additional Resources

This chapter includes the following information:

- book conventions
- JMP documentation
- JMP Help
- additional resources, such as the following:
  - other JMP documentation
  - tutorials
  - indexes
  - Web resources

Figure 1.1 The JMP Help Home Window on Windows



Contents

Formatting Conventions

The following 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 font.
- Code appears in `Lucida Sans Typewriter` font.
- Code output appears in *`Lucida Sans Typewriter`* italic font and is indented farther than the preceding code.
- **Helvetica bold** formatting 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
  - script output
- Features that are for JMP Pro only are noted with the JMP Pro icon  . For an overview of JMP Pro features, visit http://www.jmp.com/software/pro/.

Note: Special information and limitations appear within a Note.

Tip: Helpful information appears within a Tip.

JMP Documentation

JMP offers documentation in various formats, from print books and Portable Document Format (PDF) to electronic books (e-books).

- Open the PDF versions from the **Help > Books** menu or from the JMP online Help footers.
- All books are also combined into one PDF file, called *JMP Documentation Library,* for convenient searching. Open the *JMP Documentation Library* PDF file from the **Help > Books** menu.
- e-books are available at online retailers. Visit http://www.jmp.com/support/downloads/documentation.shtml for details.
- You can also purchase printed documentation on the SAS website:

  http://support.sas.com/documentation/onlinedoc/jmp/index.html

JMP Documentation Library

The following table describes the purpose and content of each book 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.
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.
Basic Analysis	Perform basic analysis using this document.	Describes these Analyze menu platforms: • Distribution • Fit Y by X • Matched Pairs • Tabulate How to approximate sampling distributions using bootstrapping and modeling utilities are also included.

Document Title	Document Purpose	Document Content
Essential Graphing	Find the ideal graph for your data.	Describes these Graph menu platforms: • Graph Builder • Overlay Plot • Scatterplot 3D • Contour Plot • Bubble Plot • Parallel Plot • Cell Plot • Treemap • Scatterplot Matrix • Ternary Plot • Chart 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.
Design of Experiments Guide	Learn how to design experiments and determine appropriate sample sizes.	Covers all topics in the **DOE** menu and the Screening menu item in the Analyze > Modeling menu.

Document Title	Document Purpose	Document Content
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
Specialized Models	Learn about additional modeling techniques.	Describes these Analyze > Modeling menu platforms: • Partition • Neural • Model Comparison • Nonlinear • Gaussian Process • Time Series • Response Screening The Screening platform in the Analyze > Modeling menu is described in *Design of Experiments Guide*.
Multivariate Methods	Read about techniques for analyzing several variables simultaneously.	Describes these Analyze > Multivariate Methods menu platforms: • Multivariate • Cluster • Principal Components • Discriminant • Partial Least Squares

Document Title	Document Purpose	Document Content
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 • Pareto Plot • Diagram
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 • Recurrence Analysis • Degradation and Destructive Degradation • Reliability Forecast • Reliability Growth • Reliability Block Diagram • 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 • Multiple Correspondence Analysis • Factor Analysis • Choice • Uplift • Item Analysis

Document Title	Document Purpose	Document Content
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.

Note: The **Books** menu also contains two reference cards that can be printed: The *Menu Card* describes JMP menus, and the *Quick Reference* describes JMP keyboard shortcuts.

JMP Help

JMP Help is an abbreviated version of the documentation library that provides targeted information. You can open JMP Help in several ways:

- On Windows, press the F1 key to open the Help system window.
- 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.
- Search and view JMP Help on Windows using the **Help > Help Contents**, **Search Help**, and **Help Index** options. On Mac, select **Help > JMP Help**.
- Search the Help at http://jmp.com/support/help/ (English only).

Additional Resources for Learning JMP

In addition to JMP documentation and JMP Help, you can also learn about JMP using the following resources:

- Tutorials (see "Tutorials" on page 29)
- Sample data (see "Sample Data Tables" on page 29)
- Indexes (see "Learn about Statistical and JSL Terms" on page 29)

- Tip of the Day (see "Learn JMP Tips and Tricks" on page 30)
- Web resources (see "JMP User Community" on page 30)
- JMPer Cable technical publication (see "JMPer Cable" on page 30)
- Books about JMP (see "JMP Books by Users" on page 31)
- JMP Starter (see "The JMP Starter Window" on page 31)

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, then 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 creating a pie chart, using Graph Builder, and so on.

Sample Data Tables

All of the examples in the JMP documentation suite use sample data. Select **Help > Sample Data Library** to do the following actions 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\<version_number>\Samples\Data

On Macintosh: \Library\Application Support\JMP\<version_number>\Samples\Data

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

Learn about Statistical and JSL Terms

The **Help** menu contains the following indexes:

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.

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. See the *Using JMP* book for details.

Tooltips

JMP provides descriptive tooltips when you place your cursor 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: You can hide tooltips in the JMP Preferences. Select **File > Preferences > General** (or **JMP > Preferences > General** on Macintosh) and then deselect **Show menu tips**.

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**.

JMPer Cable

The JMPer Cable is a yearly technical publication targeted to users of JMP. The JMPer Cable is available on the JMP website:

http://www.jmp.com/about/newsletters/jmpercable/

JMP Books by Users

Additional books about using JMP that are written by JMP users are available on the JMP website:

http://www.jmp.com/support/books.shtml

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.

- To open the JMP Starter window, select **View** (**Window** on the Macintosh) **> 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 Macintosh, select **JMP > Preferences > Initial JMP Starter Window**.

Chapter 2

Introduction

Welcome to the JMP Scripting Language

The JMP Scripting Language, or *JSL*, lets you write scripts to recreate 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 any time 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* book begins with basic information (such as terminology and syntax) for JMP users who are not familiar with the scripting language. The book 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 Excel.
Chapter 15	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 16	Contains a collection of recipes, or script examples, that you can copy and modify for your own use.
Appendices A and B	Provides information about compatibility issues with the previous version of JMP 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.

The Scripting Index window includes the following buttons:

 Click the **Search** button to begin the search.

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

 Click the **Settings** button to set search types and parameters.

Several types of searches are available from the **Settings** button:

Partial Match returns all entries that contain at least a part of the “string” for example, a search for “leas” will return messages such as “Release Zoom” and “Partial Least Squares”. This option is the default search type.

Exact Phrase returns entries that contain the exact string, for example, a search for "text" will return all elements that contain the "text" string.

All Terms returns entries that contain either or both strings, for example, a search for "t test" will return all elements that contain either or both of the search strings, "Pat Test", "Shortest Edit Script" and "Paired t test".

Any Term returns entries that contain either of the search strings, for example, a search for "text string" returns "Context Box", "Drag Text", and "Is String".

Regular Expression allows you to use the wildcard (*) and period (.) in the search box, for example, searching for "get *name" returns messages such as "Get Name Info" and "Get Namespace". Searching for "get.*name" returns items such as "Get Color Theme Names", "Get Name Info", and "Get Effect Names".

Several search parameters are also available from the **Settings** button:

All Fields specifies that JMP search all fields in the index for the search string.

Titles Only specifies that JMP search only index titles for the search string.

Examples Only specifies that JMP search only index examples for the search string.

Without Examples specifies that JMP exclude examples from the search.

Click an item's **Topic Help** button to read more about the item in JMP's online Help system.

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 **Script > Save Script to Script Window** from the red triangle menu for the report. The following script appears:

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

```
);
```

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 this book.

Operators and Functions

An *operator* is one- or two-character symbol (such as + or =) for common arithmetic actions.

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;
Add( 2, 3 );
```

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. This book commonly uses `dt` to represent data table

references. Here are some of the more common names used in this book 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 this book, 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.

Expressions

An *expression* is a section of JSL code that accomplishes a task. JSL expressions hold data, manipulate data, and send commands to objects. For example, the following expression opens the Big Class.jmp sample data table and creates a Bivariate graph:

```
Open( "$SAMPLE_DATA/Big Class.JMP" );
  Bivariate( Y( :weight ), X( :height ) );
```

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. This book uses the script editor's default formatting for capitalization, spaces, returns, tabs, and so on. See "Using the Script Editor" on page 55 in the "Scripting Tools" chapter for details 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, with message contents 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 97 in the "JSL Building Blocks" chapter for more information.

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

```
New Window( "A Window",
   << modal,
   Text box( "Hello, World" ),
   Text Box( "-----" ),
   ButtonBox( "OK" )
);
```

Notice the following:

- Names can have embedded spaces. See “Names” on page 88 in the “JSL Building Blocks” chapter for more information.
- Message contents are enclosed in parentheses, which must be balanced. See “Parentheses” on page 85 in the “JSL Building Blocks” chapter.
- Items are separated by commas. See “Commas” on page 85 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.

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 Excel files, and producing reports. A final tutorial shows you how to put it all together into a single script to open an Excel file and produce three reports automatically.

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

Contents

Capturing a Script for an Analysis Report

The basic steps for capturing a script to reproduce an analysis are as follows:

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 the 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**.

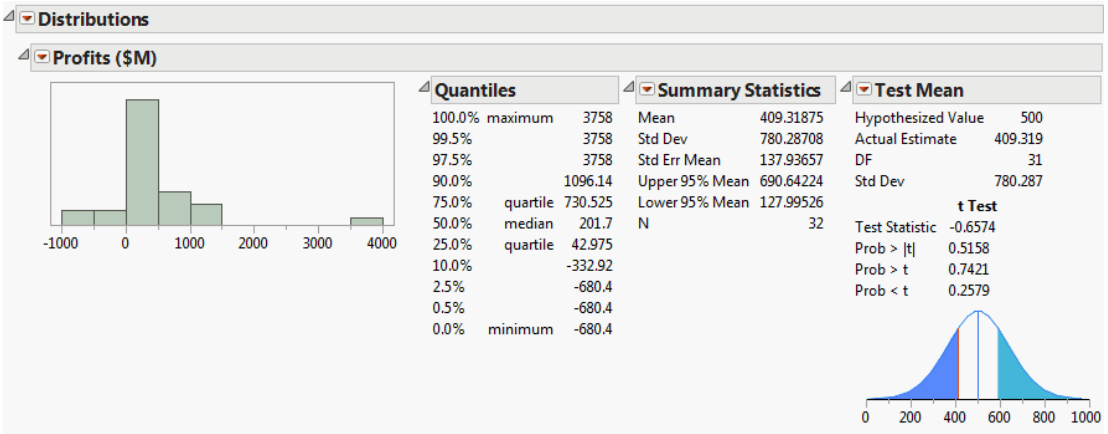
   The Distribution report window appears.
5. From the red triangle menu next to Distributions, select **Stack** to make your report horizontal.
6. From the red triangle menu next to Profits ($M), deselect **Outlier Box Plot** to turn the option off.
7. From the red triangle menu next to Profits ($M), select **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.

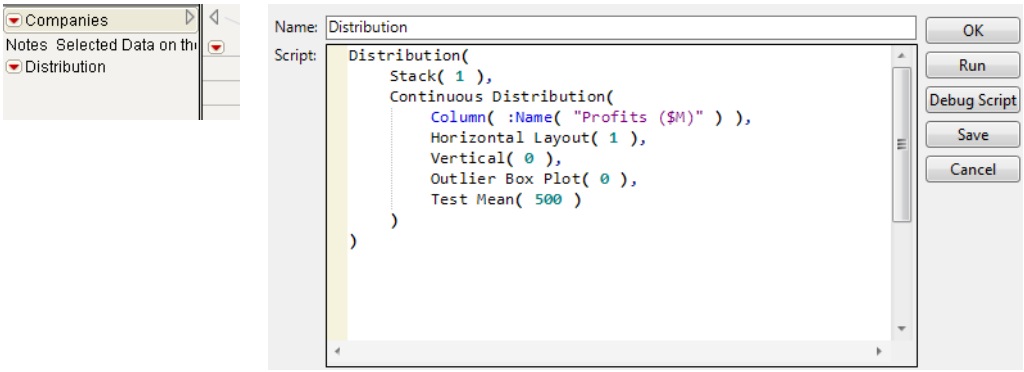
Figure 3.1 Customized Distribution Report



10. From the red triangle menu next to Distributions, select **Script > Save Script to Data Table**.

    Your data table now has a script named Distribution saved to it. From the red triangle menu for the script, 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, select **Run Script** from the red triangle menu for the script.

Capturing a Script for a Data Table

The basic steps for capturing a script to reproduce a data table are as follows:

1. Open the data table.
2. Make any changes that you need. For example, add a script, correct values, add new columns.

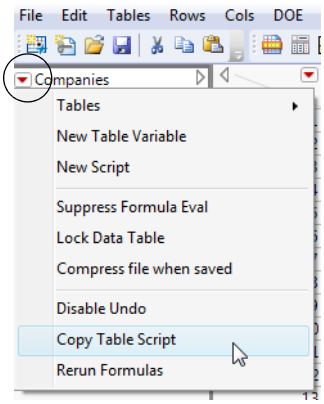
3. Capture the script to recreate your data table.

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, using 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. From the red triangle menu for Source, 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 126 in the "Types of Data" chapter.

Gluing Scripts Together

Suppose new data is saved out to an 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 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.xls sample import data file. The file is located in JMP's Samples/Import Data folder.

   ```
   Open( "$SAMPLE_IMPORT_DATA/Solubil.xls" );
   ```

   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 Excel file by selecting **Edit > Run Script**.

   The Excel file opens as a data table.

Note: 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 126 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. Hold down CTRL and select **Histogram Options > Show Counts** from the red triangle menu for eth.

   Bar counts are added to all six histograms.
5. In the Distribution window, select **Script > Copy Script** from the red triangle menu next to Distributions.
6. Place your cursor in the script window 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.

Figure 3.4 The Completed Script

```
Open ("$SAMPLE_IMPORT_DATA/Solubil.xls");
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" ),
    Matrix Format( "Square" ),
    Scatterplot Matrix(
        Density Ellipses( 1 ),
        Shaded Ellipses( 0 ),
        Ellipse Color( 3 )
    )
);
```

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 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 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 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. Hold the CTRL key while you select the JSL file and 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**.

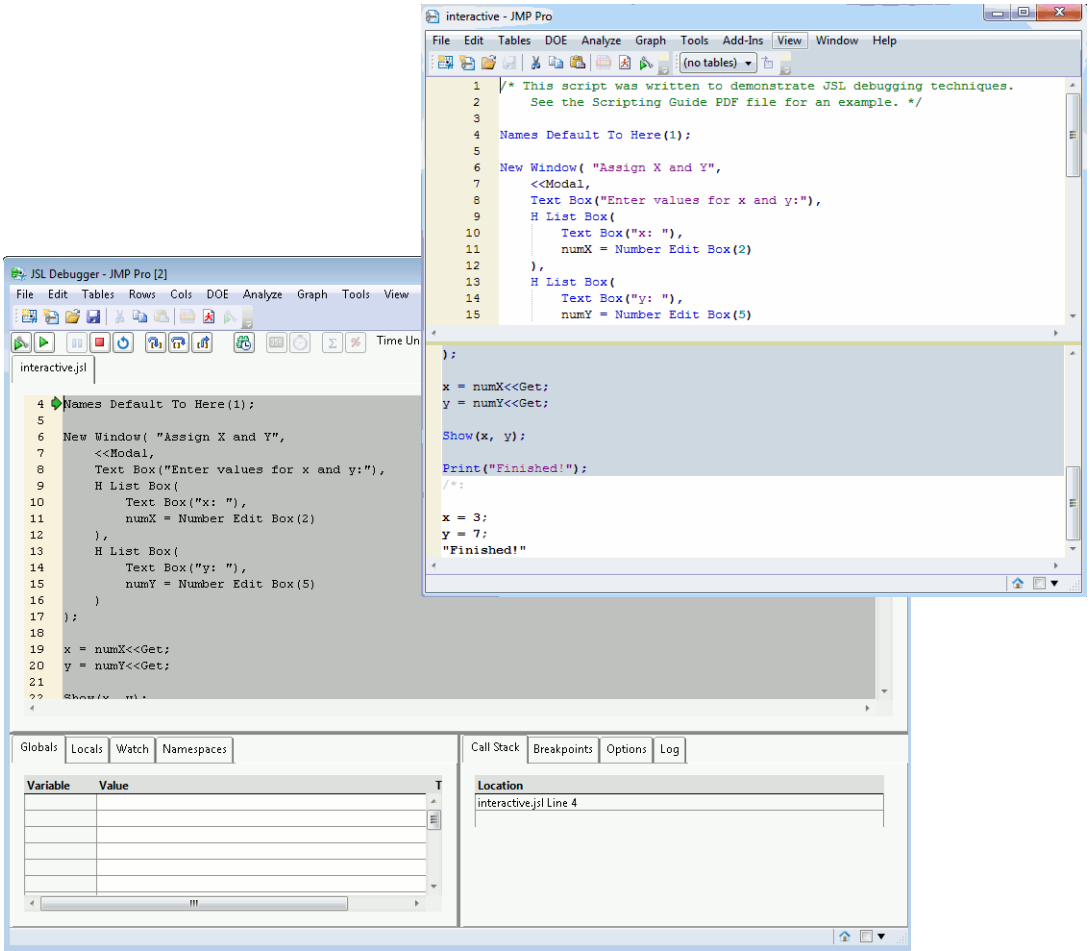
Chapter 4

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



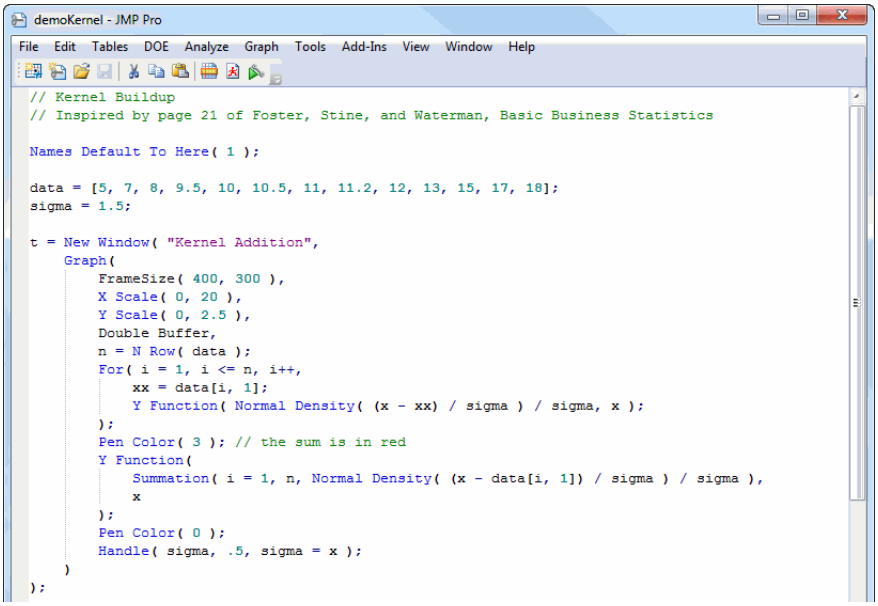
Contents

Using 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



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, hold down the Control key, 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 Macintosh). You can also select **Edit > Stop Script**. On Macintosh, **Edit > Stop Script** is available only when the script is running.

Edit a Script

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

Color Coding

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 Debugger background, disabled background, and guides
- green for comments
- purple for strings
- teal blue for numbers
- dark blue for operator symbols, the first keyword, and JSL objects
- medium blue for operator names, second and third keywords, and macros
- red for unknown objects

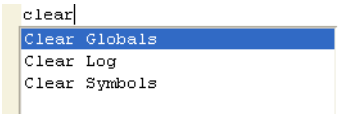
Customize colors in the preferences. See "Set Preferences for the Script Editor" on page 63.

Auto Complete Functions

If you do not remember the exact name of a function, use auto completion 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 Macintosh).

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

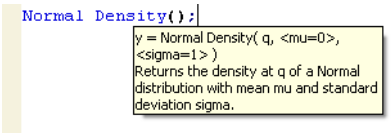


Tooltips

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**.

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.

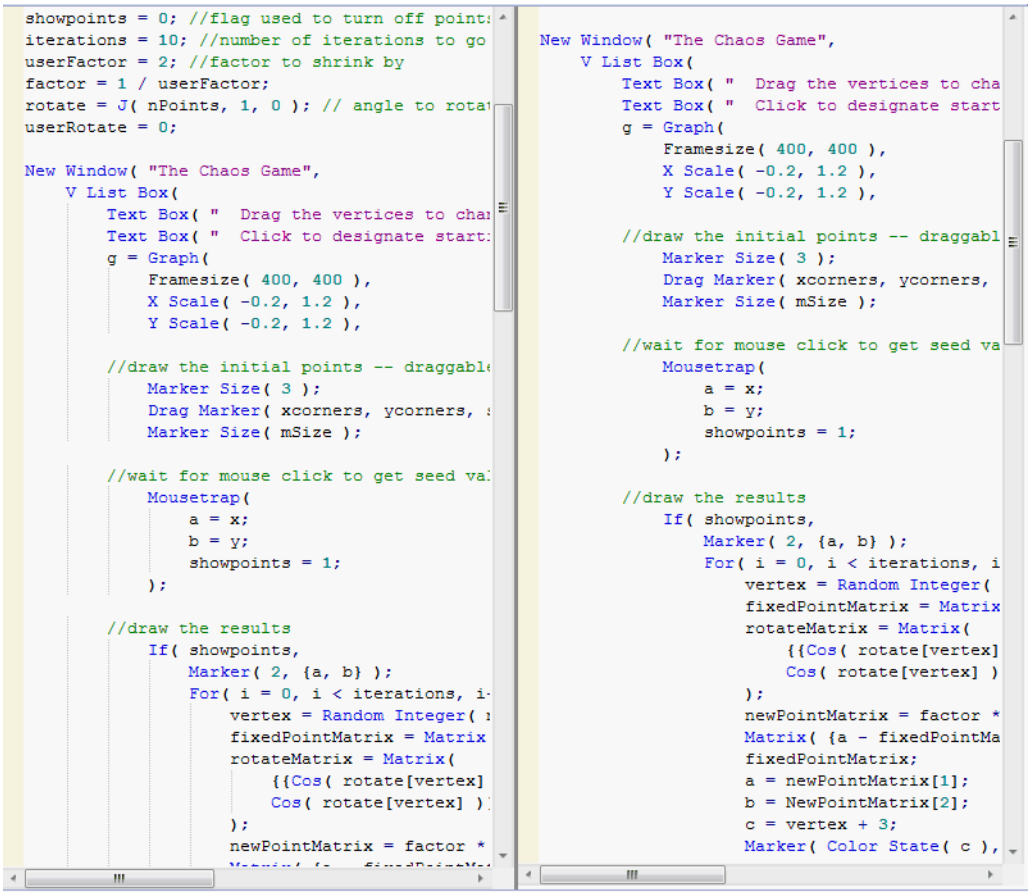
   A tooltip shows the name of the variable and its value: “eight”.

Split a Window

You can split the Script Editor window into two vertical or horizontal windows. This feature allows 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 back to a single window, right-click and select **Remove Split**.

Figure 4.5 Example of Splitting a Window Horizontally

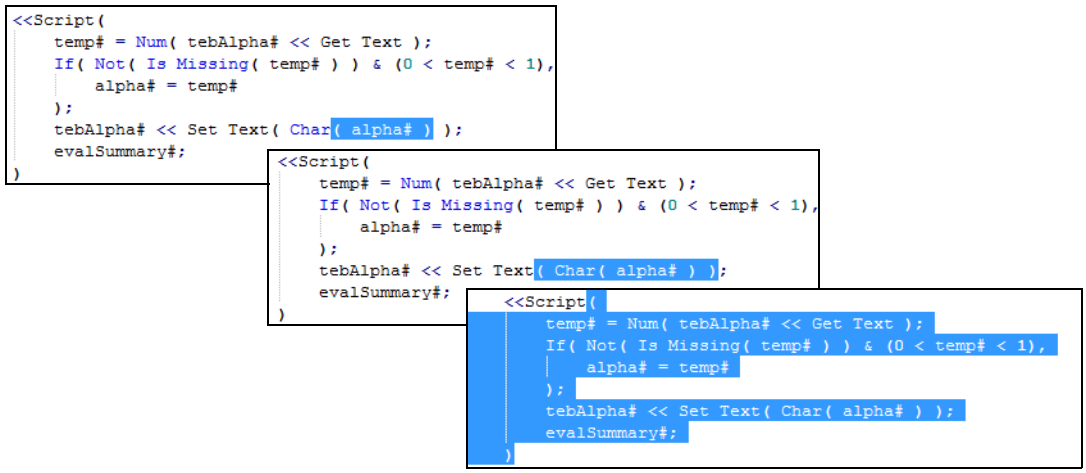


Match Parentheses, Brackets, and Braces

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

- The matching closing fence is added when you type an opening fence.
- When you place your cursor 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 Macintosh), 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



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 auto completion of braces in the JMP preferences. See "Set Preferences for the Script Editor" on page 63 for details.

Select a Rectangular Block of Text

To select a rectangular block of text, hold down the ALT key 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, hold down the Ctrl key on Windows or Command key on Mac 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 Macintosh, the text is copied by default.

Drag and drop text as follows:

- 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 the *Using JMP* book for details 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).

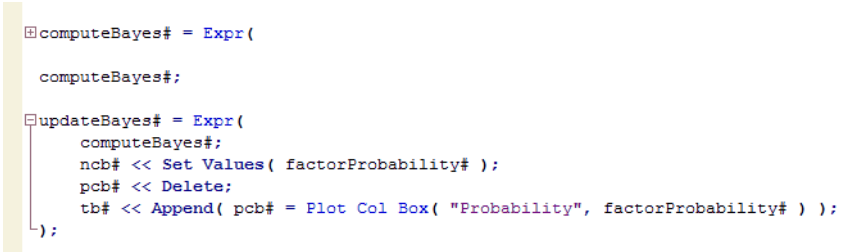
Add Code Folding Markers

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

To turn on this feature, select **JSL 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**.

After you select this preference, `Function` and `Expr` expressions are foldable. See “Add More Folding Keywords” on page 62 for details about adding folding markers to other expressions.

Figure 4.7 Code Folding Markers Shown in a Script



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.

Note: Path names in this section refer to the JMP folder. In JMP Pro, the folder is named "JMPPro". In JMP Shrinkwrap, the folder is named "JMPSW".

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

- C:\ProgramData\SAS\JMP\<version>\
- C:\ProgramData\SAS\JMP\
- C:\Users\<user>\AppData\Roaming\SAS\JMP\<version>\
- C:\Users\<user>\AppData\Roaming\SAS\JMP\
- C:\Users\<user>\Documents\

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

- /Library/Application Support/JMP/<version>/
- /Library/Application Support/JMP/
- ~/Library/Application Support/JMP/<version>/
- ~/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:

Option	Description
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

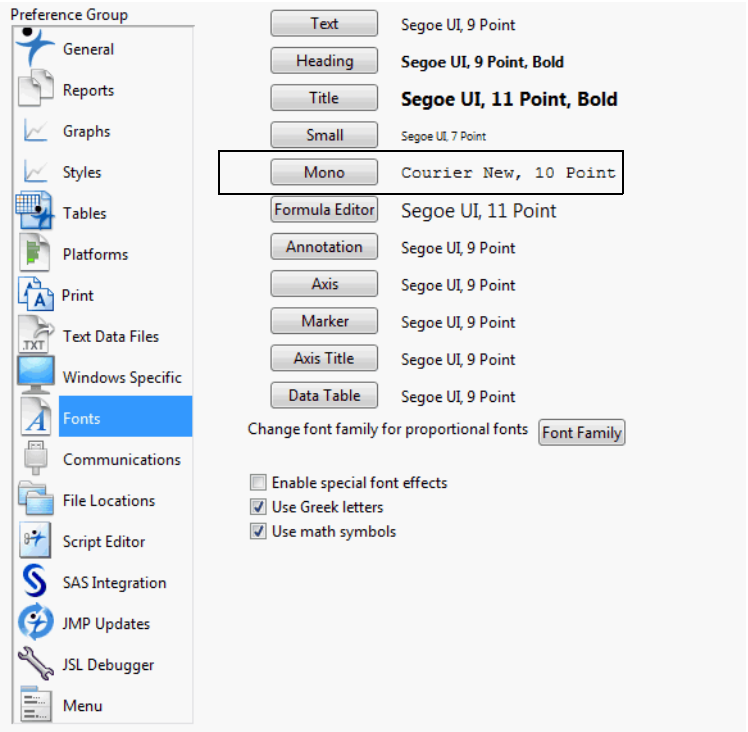
In the JMP preferences, customize the script editor settings such as the font, colors, and spacing options.

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 details about font preferences, see the *Using JMP* book.

Figure 4.8 Changing the Font Properties for Script Windows



Set Script Editor Preferences

Select **File > Preferences > Script Editor** to further customize the editor.

Preference	Description
Use tabs	Select this option to enable tabs in your scripts. This option is selected by default. Clear this option to replace any tab that you type with spaces.
Tab width	Enter how many spaces a tab should indent. If you have disabled tabs, any tab you type is replaced with the number of spaces specified. The default value is 4.
Extra space at bottom of document	Select this option to enable scrolling up from the last blank lines of a script. This option is selected by default on Windows and deselected on Macintosh.

Preference	Description
Auto-complete parentheses and braces	Select this option to enable the script editor to automatically add closing parentheses, square brackets, and curly braces when you type an opening one. This option is selected by default.
Show line numbers	Select this option to show the line numbers on the left side of the script editor. This option is cleared by default.
Show indentation guides	Select this option to see faint vertical lines that mark indention. This option is selected by default.
Show operator tips	Select this option to see tooltips for JSL operators. This option is selected by default.
Show variable value tips	Select this option to see tooltips for variable values. This option is selected by default.
Wrap Text	Select this option to always wrap text in the script editor. This option is off by default.
Show embedded log on script window open	Select this option to have an embedded log window appear in the scripting window when editing or running scripts. This option is deselected by default.
Color unknown object messages	Select this option to cause the script editor to color object messages that are not defined for the target object. Unknown object messages will appear in the color specified by **Message unknown color**. **Note:** This option can affect the performance of the JSL editor due to the amount of effort to look up message names in context.
Save and restore document state information	Saves the state of collapsed and expanded code, and restores that state when the script is reopened.
Spaces inside parentheses	Select this option to cause the script editor to add spaces between parentheses, brackets, and braces and their contents for automatically formatted scripts. This is on by default.
Spaces in operator names	Select this option to cause the script editor to add spaces between words within operator names. For example, turning on this option results in `New Window` instead of `NewWindow`. This option is selected by default.

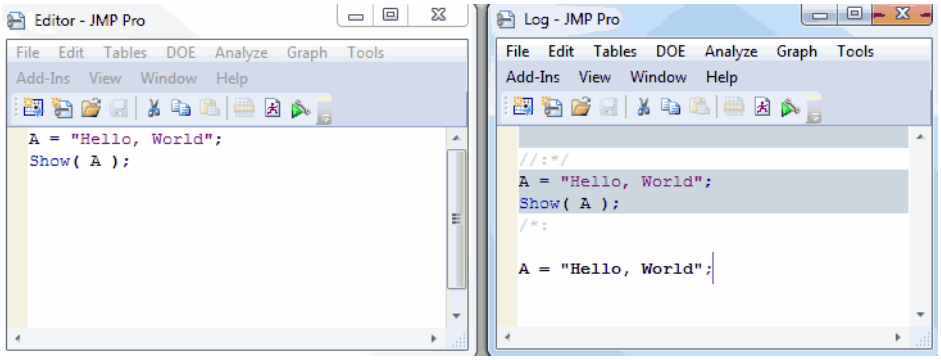
Preference	Description
JSL code folding	Select this option to use code folding markers in the script editor, which mark the opening and closing of `Function()` and `Expr()` expressions. You can expand and collapse these marked blocks of code. This option is off by default. You can also choose the appearance of the marker using the **JSL code folding marker** menu. Select **Allow additional code folding keywords** to enable using additional keywords for folding markers in the script editor. See "Add More Folding Keywords" on page 62 for details.
Color selection	To set your own color for any of the listed types, click the color box and select your color. See "Color Coding" on page 56 for details on default settings.

For more details about script editor preferences, see the *Using JMP* book.

Working with the Log

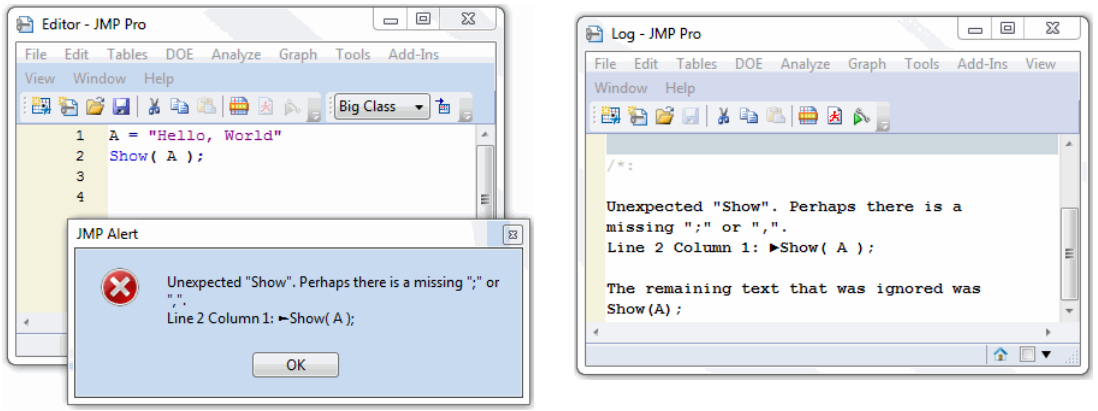
As you run a script, the output appears in the log window. The actual script is shaded in the log, and the output appears beneath it (Figure 4.9).

Figure 4.9 The Script Window (left) and the Log Window (right)



Syntax and compatibility errors are reported in the log, including the line number and code that JMP could not process (Figure 4.10).

Figure 4.10 Syntax Error Message



Open the log by selecting **View > Log** (**Window > Log** on Macintosh).

Tips:

- On Windows, you can control when the log opens: when JMP starts, when text is written to the log, or only when you open it. Select **File > Preferences > Windows Specific** to change the Open the JMP Log window setting.
- To omit compatibility warnings from the log, deselect **Show log warnings for JSL compatibility changes in 12** in the JMP General preferences.

Show the Log in the Script Window

You can view the log inside the script window by right-clicking and selecting **Show Embedded Log**. 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 always appears in the Scripting Index script window but is not available in Application Builder and the Debugger.

Save the Log

You can also save logs as text files, which can be viewed with any text editor. Double-clicking a log text file opens it in your default text editor, not in JMP.

1. Make the log window active (click the Log window to make it the front-most window).
2. From the **File** menu, select **Save** or **Save As.**
3. Specify a filename, including the extension .txt on Windows. On Macintosh you cannot save a log as a .txt file. .jsl is appended to the file name.
4. Click **Save.**

Debug or Profile Scripts

In an open script, click the **Debug Script** button (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 (Macintosh).

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.

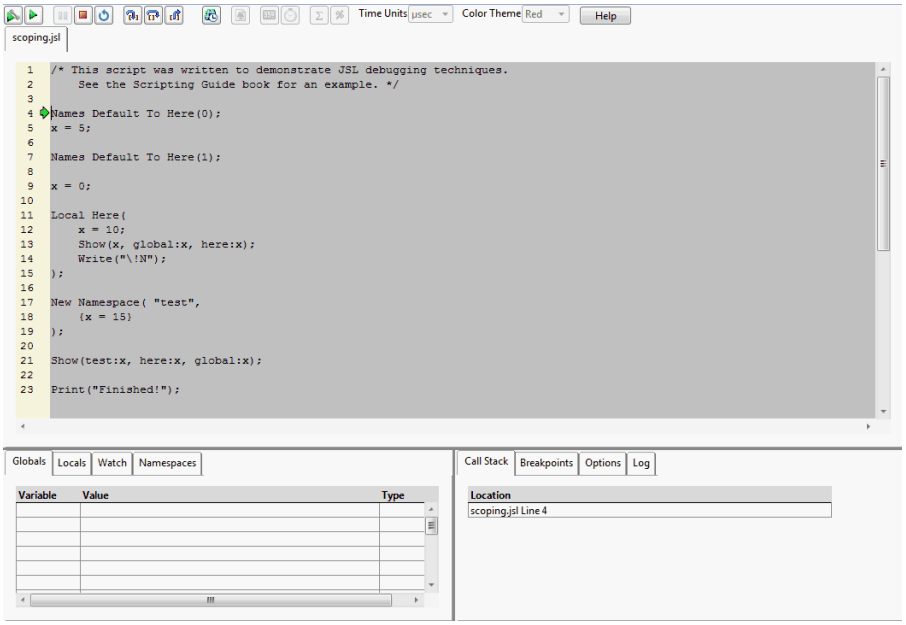
Tip: To debug a script automatically when you open it, include `Run JSL(1)` in the `Open()` statement:

```
Open( "$SAMPLE_SCRIPTS/scoping.jsl", Run 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.

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.
	Run without breakpoints Run profiler	Runs the script through the end without stopping.

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

Button	Button Name	Action
	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 may not be able to break execution if the executing script is waiting on some interactive user action, such as completing a dialog 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.
	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. Note the following: • 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.

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

Button	Button Name	Action
	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  .
	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 75.

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 75.

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 hard to watch in the Globals or Locals lists. See "Work with Watches" on page 75.

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 75.

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. See "Work with Breakpoints" on page 72. You can also double-click a row on the Breakpoints tab to move the cursor to the specified line.

Options Set the Debugger preferences interactively on this tab. See "Modify Preferences in Debugger" on page 76.

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 you can look at the values of variables or start stepping line by line.

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, and then enter the breakpoint information.

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

- In the Debugger margin, click on 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 recreating 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 only want to see values 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.

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.

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: "/C:/Users/<username>/AppData/Roaming/SAS/JMP/<version number>/"
- Macintosh: "/Users/<username>/Library/Application Support/JMP/<version number>/"

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". In JMP Shrinkwrap, the path refers to "JMPSW".

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

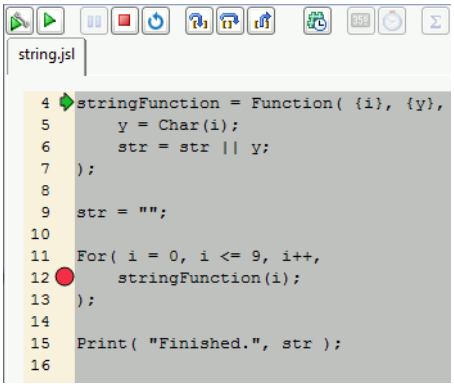
The following example shows how to set a breakpoint in a loop and watch variables change through each iteration of the loop.

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 (Figure 4.12).

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

   ```
   stringFunction(i);
   ```

Figure 4.12 Set the Breakpoint



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.

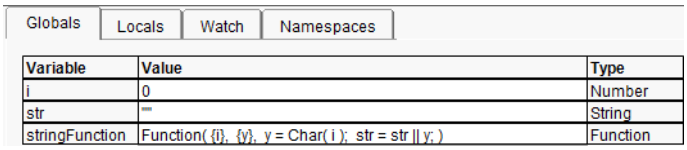
   – `i` has been assigned to 0.

   – `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



4. Click **Run** again.

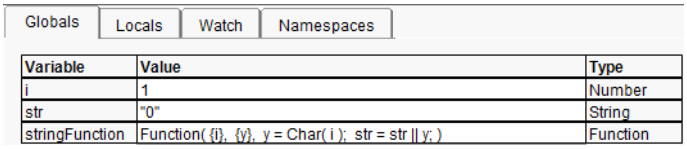
   The script runs until it hits the breakpoint. The results are shown in Figure 4.14.

   – `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



5. Click **Run** again.

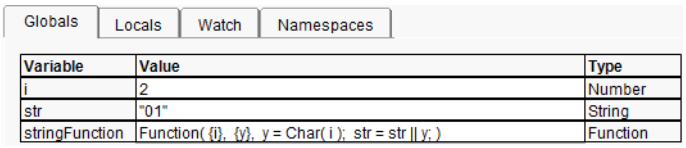
   The script runs until it hits a breakpoint. The results are shown in Figure 4.15.

   – `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



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.

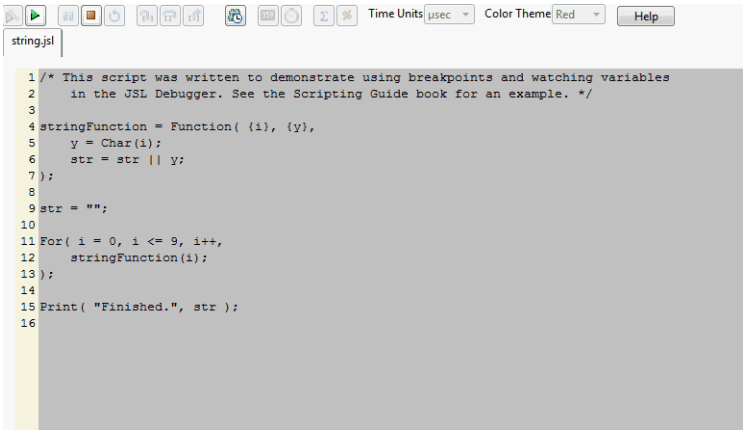
Using the JSL Profiler

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 .
3. Click the **Profile JSL Script** button .

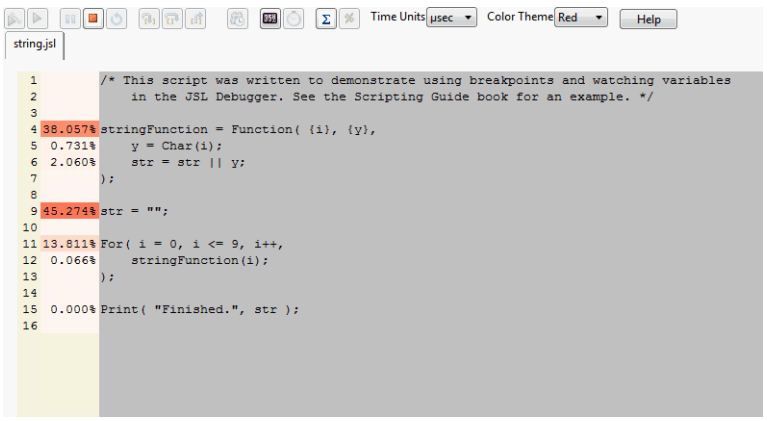
Figure 4.16 Initial JSL Profiler Window



4. Click the **Run** button to start profiling.

   The profiler collects information on 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



In the left margin, the selected statistics are displayed. Percent of time is displayed by default. Click the **Show Profile by Count** button Σ to switch to percent of statement counts instead. The left margin is color-coded to allow for quick identification of problematic performance areas.

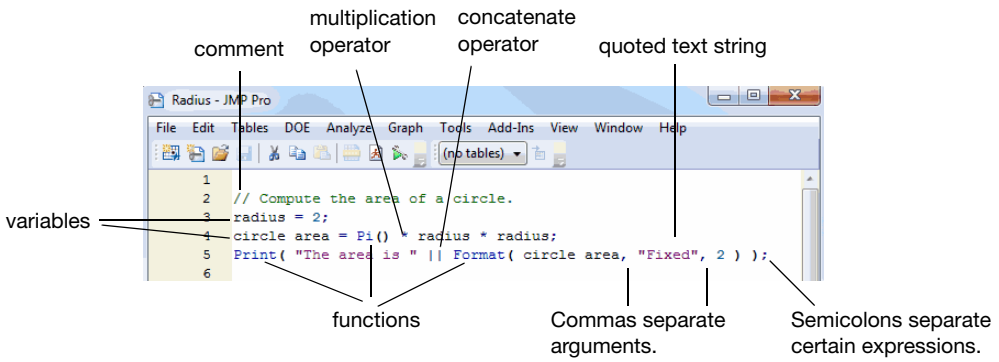
Chapter 5

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);
If(Y < 20, X = Y; Z < 3, A);
TableBox(
     stringColBox("Age",a),
     NumberColBox("Count",c),...)
```

Note: To glue a sequence of commands into a single argument inside a function, separate each sequence with a semicolon. For more information, see "Semicolons" on page 86.

Parentheses

Parentheses have several purposes in JSL:

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

```
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 Macintosh) 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 59 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 details about separating expressions with semicolons, see "Alternatives for Gluing Expressions Together" on page 104.

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 90 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 Macintosh, 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 datetime 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 130 in the "Types of Data" chapter. See "Currency" on page 142 in the "Types of Data" chapter for details 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 ⊆)
  - 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 placed inside a special parser directive called `Name()`. For example, to use a global variable with the name `taxable income(2011)`, you must use `Name()` every time the variable appears in a script:

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

`Name()` is harmless when it is not needed. For example, `tax` and `Name("tax")` are equivalent.

For more information about how JMP interprets names, see “Rules for Name Resolution” on page 97.

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

Operators are one- and two-character symbols for common arithmetic actions. 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`)

JSL operators all have function equivalents.

To make writing algebraic expressions easier, JSL uses certain special character operators. 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.
++	Post Increment	*a*++ Post Increment(*a*)	Adds one (1) to *a*, in place.
––	Post Decrement	*a*–– Post Decrement(*a*)	Subtracts one (1) from *a*, in place.
^	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*.

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

Operator		Function Syntax	Explanation
!	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*.)
+	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 172 in the "Data Structures" chapter or the *JSL Syntax Reference* for details.
|/	VConcat	*matrix1*|/*matrix2* VConcat(*matrix1*, *matrix2*)	Vertically concatenate matrices. (Use || or Concat() to horizontally concatenate matrices.)

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

Operator		Function Syntax	Explanation
`::`	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 98 for details.)
`<<`	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 117, 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*)	
`<=, <`	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 117, for treatment of missing values.

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

Operator		Function Syntax	Explanation
|	Or	*a*|*b* Or(*a*,*b*)	Logical Or. Returns true if either or both are true. See "Missing Values" on page 117, 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*.

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. "Advanced Scoping and Namespaces" on page 237 provides more details.

Note: The `Names Default to Here` option 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.

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. For more information, see "Rules for Name Resolution" on page 97.

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 68 in the "Scripting Tools" chapter for more information.

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 97 for details 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`.

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.

Resolving Unscoped Names

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

1. When the name is part of a script for an object, it is usually the name of an option or method in the object. For example, `Show Points()` is an option in the Bivariate object:

   ```
   obj = Bivariate(y(weight), x(height));
   obj << Show Points(1);
   ```

2. If a name is not preceded by the `:` scoping operator, look it up in a namespace, such as Global or Here.
3. If a name is followed by a pair of parentheses `()`, look up the name as a built-in function (not a user-defined function).

4. If a name is preceded by the `:` scoping operator, look it up as a data table column or table variable.
5. If a name is preceded by the `::` scoping operator, look it up as a global variable.
6. Look it up as a local variable.
7. Look it up as a platform launch name (for example, `Distribution` or `Bivariate`).
8. If a name is used as the left side of an assignment (the L-value) and `Names Default To Here(0)` is at the top of the script, create and use a global variable.

Exception

- 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( "$SAMPLE_DATA/Big Class.jmp" );
Show( weight << Get As Matrix ); // weight  resolves to a column name
Close( dt, NoSave );
Show( weight << Get As Matrix ); // weight  cannot be resolved
/* Reopen the data table */
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
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( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( dt, 1 ); // col is Column( "weight" )
Close( dt, NoSave );
/* Reopen the data table */
dt = Open( "$SAMPLE_DATA/Big Class.jmp " );
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 237 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 239 in the "Programming Methods" chapter for details.

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 99 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");
```

Preventing Column Name and Variable Name Conflicts

When you run a script that includes a column and variable with the same name, an `Invalid Row Number` error occurs. To prevent this problem, use unique column and variable names, or scope the names as follows:

- 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. For details, see "Scoped Names" on page 240 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).

If the data table has a table variable by that name, then the table variable takes precedence. In all other cases, it binds to a global, local, or argument. For more information about global and local variables, see "Global and Local Variables" on page 94.

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.

```
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 366 in the "Data Tables" chapter.)

Troubleshooting 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];
   As Name( "height" )[/*###*/3];
```

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];
  ```

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

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

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

Troubleshooting 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 237 in the "Programming Methods" chapter for more information.

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 as follows:

```
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 110. See "Data Tables" chapter on page 275 for more information about working with data tables.

Will a Scoping Operator "Stick" to its Name?

Yes. Once you use a scoping operator with a name, that name continues to be resolved accordingly for all subsequent occurrences of the name. For example, a script might contain a column and a variable named age. When you declare the global variable age with the scoping operator :: at the beginning of the script, age is always interpreted as a global variable in the script. The values in the age column are not affected by the variable.

```
::age=70;
Open("$SAMPLE_DATA/Big Class.jmp");
age=5; // age is a global variable.
Show(age); // age is still a global variable.
```

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 110 for an example. "Additional Ways to Access Data Values" on page 362 in the "Data Tables" chapter also describes iterating through table rows.

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.

The basic syntax for `For()` is as follows:

```
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 Macintosh). You can also select **Edit > Stop Script**. On Macintosh, **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, `Which()` 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 Macintosh). You can also select **Edit > Stop Script**. On Macintosh, **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` equals 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 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()`, and `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".

```
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:

```
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.

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:

```
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 110. 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 are not what you want.

The following script changes the column's data type from numeric to character based on the first cell's data type. The first value, "12", is replaced with "Twelve", and the remaining cells are filled with "Other".

```
dt=open("$SAMPLE_DATA/Big Class.jmp");
For Each Row(age =
   Match(age, 12, "Twelve", 13, "Thirteen", 14, "Fourteen", 15, "Fifteen", 16,
   "Sixteen", "Other");
);
```

When data consists of integers such as 1, 2, and 3, you can save even more typing by using `Choose()`. See "Choose" on page 113 for more information.

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 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,
   "Low",
   "Medium",
   "High",
   "Unknown"
    );
);
Show(x);
```

The script works like this:

`x =`	Creates the `x` variable.
`Choose(`	Begins the `Choose()` loop.
`group,`	Evaluates the value of group.
`"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, "Low", group==2, "Medium", group==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 finds the `y` value corresponding to a given `x` value between two points (`x1`, `y1` and `x2`, `y2`). A linear interpolation is applied to the values. You might use `Interpolate()` to calculate missing values between data points.

The data points can be specified as a list:

```
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 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.

Notes:

- The data points in each list or matrix must create a positive slope. 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 for discrete data. See "Step" on page 115 for details.

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 90. The sections that follow the table provide more details about comparison and logical operators.

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 == "".

Table 5.5 Some Special-Case Comparison Tests *(Continued)*

Test	Result	Explanation
`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.
`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. For example, the following script:

```
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"
                )
             )
          ) );
```

returns the following results:

```
"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.

The two exceptions to this rule involve comparing a missing value to a known value:

- If one value is true and another is missing, `Or()` returns true. (Only one value in an `Or()` test needs to be true to get a true result.)
- If one value is false and another is missing, `And()` returns false. (Both values in an `And()` test must be true to get a true result.)

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 as follows:

```
If( :A, "true", Is Missing( :A ), "missing", "false" );
```

`Is Missing( :A )` returns “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 operators (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));
   1
Show(Is Scriptable(col));
   1
Show(Is Scriptable(plat));
   1
Show(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. 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, not its result" on page 223 for details.
`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 Functions chapter in the *JSL Syntax Reference* book 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.

For example, if the operating system is Windows, the following script loads a Windows Dynamic Link Library (DLL):

```
If(Host is("Windows"),
  dll_obj = Load DLL("C:/Windows/System32/user32.dll")
);
```

You could also use `Host Is()` to specify text sizes in reports for different operating systems. If you commonly write your scripts on Windows and share them with Macintosh users, the results can look different from what you intended. For example, the following expression sets the text to a larger size on Macintosh and a smaller size 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 "12.0.0" in JMP 12
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 `<<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
```

Unicode Characters

JMP supports both Unicode UTF-8 and UTF-16 standards for encoding and representing text for most of the world languages. Refer to the The Unicode Consortium for code charts and details 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.

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	The second argument in `Register Addin()` is assigned to this variable. See the `Register Addin()` section of the *JSL Syntax Reference* for details. When you create an add-in through Add-In Builder, the `$ADDIN_HOME` definition is based on your computer's operating system: • Windows: "C:/Users/<username>/AppData/Roaming/SAS/JMP/Addins/" • Macintosh: "/Users/<username>/Library/Application Support/JMP/Addins/"

Table 6.1 Predefined Path Variable Definitions *(Continued)*

Variable	Path
ALL_HOME	• Windows (JMP): "/C:/ProgramData/SAS/JMP/<version number>/" • Windows (JMP Pro): "/C:/ProgramData/SAS/JMPPro/<version number>/" • Windows (JMP Shrinkwrap): "/C:/ProgramData/SAS/JMPSW/<version number>/" • Macintosh: "/Library/Application Support/JMP/<version number>/"
DESKTOP	• Windows: "/C:/Users/<username>/Desktop/" • Macintosh "/Users/<username>/Desktop/"
DOCUMENTS	• Windows: "/C:/Users/<username>/Documents/" • Macintosh: "/Users/<username>/Documents/"
GENOMICS_HOME	"/<JMP Genomics installation directory>/"
HOME	• Windows (JMP): "C:/Users/<username>/AppData/Roaming/SAS/JMP/<version number>/” • Windows (JMP Pro): "/C:/ProgramData/SAS/JMPPro/<version number>/" • Windows (JMP Shrinkwrap): "/C:/ProgramData/SAS/JMPSW/<version number>/" • Macintosh: "/Users/<username>/"
SAMPLE_APPS	• Windows: /C:/<JMP installation directory>/Samples/Apps/" • Macintosh: "/Library/Application Support/<JMP installation directory>/Samples/Apps/"
SAMPLE_DATA	• Windows: "/C:/<JMP installation directory>/Samples/Data/" • Macintosh: "/Library/Application Support/<JMP installation directory>/Samples/Data/"
SAMPLE_IMAGES	• Windows: "/C:/<JMP installation directory>/Samples/Images/" • Macintosh: "/Library/Application Support/<JMP installation directory>/Samples/Images/"

Table 6.1 Predefined Path Variable Definitions *(Continued)*

Variable	Path
SAMPLE_IMPORT_DATA	• Windows: "/C:/<JMP installation directory>/Samples/Import Data/" • Macintosh: "/Library/Application Support/<JMP installation directory>/Samples/Import Data/"
SAMPLE_SCRIPTS	• Windows: "/C:/<JMP installation directory>/Samples/Scripts/" • Macintosh: "/Library/Application Support/<JMP installation directory>/Samples/Scripts/"
TEMP	• Windows: "/C:/Users/<username>/AppData/Roaming/Temp/" • Macintosh: "/private/var/folders/.../Temporary Items/"
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/<version number>/" • Windows (JMP Pro): "/C:/ProgramData/SAS/JMPPro/<version number>/" • Windows (JMP Shrinkwrap): "/C:/ProgramData/SAS/JMPSW/<version number>/" • Macintosh: "/Users/<username>/Library/Application Support/JMP/<version number>/"

Path variable definitions are updated automatically based on the version of JMP you are using. For example, when you run a JMP 9 script in JMP 12, the JMP 12 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/12/"
```

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/11/Samples/Data/Big Class.jmp
```

Without the slash that follows `$SAMPLE_DATA`, the path is interpreted as:

```
C:/Program Files/SAS/JMP/11/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"); // returns "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:

```
Open("cleansers.jmp");
```

resolves as C:/users/smith/data/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 Macintosh. 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); //returns 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 + inDays(2));` shows as: *10Dec2000* Values that represent less than a year are returned as durations. `As Date(50000);` shows as: *:0:13:53:20*

Table 6.2 Date-Time Functions *(Continued)*

Function	Explanation
`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.
`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 137.
`Hour(datetime)`	Returns an integer representation for the hour part of the *date-time* value supplied.
`In Days(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.
`In Hours(n)`	
`In Minutes(n)`	
`In Weeks(n)`	
`In Years(n)`	
`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"`.

Table 6.2 Date-Time Functions *(Continued)*

Function	Explanation
`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.
`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 `dd`Monyyyy format.
`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()
   3388649872

Short Date(Today());
   "05/19/2011"
Long Date(Today());
   "Thursday, May 19, 2011"
Abbrev Date(Today());
   "5/19/2011"
MDYHMS(Today());
   "05/19/2011 11:37:52 AM"
```

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);
   "Thursday, May 19, 2011"
```

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

	Date	Time
1	05/24/2011	12:05:57 PM

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(
        If(row()==1, ., // returns a missing value for the first row
        (: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())/InWeeks();
  55.6559441137566
```

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())/InYears(10);
  2.65581440286967
```

Two- 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 datetime separators selected in the Region and Language control panel (Windows) or the Date & Time preferences (Macintosh) 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 Macintosh)			
	during or after 1990 (on Windows)	20__	enter 5 in year 1991	2005
	after 1990 (on Macintosh)			
11–89 (on Windows)	any time	current century	enter 13 in year 1988	1913
11–90 (on Macintosh)			enter 13 in year 2024	2013
90–99 (on Windows)	before 2011	19__	enter 99 in year 1999	1999
91–99 (on Macintosh)	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 123.

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.

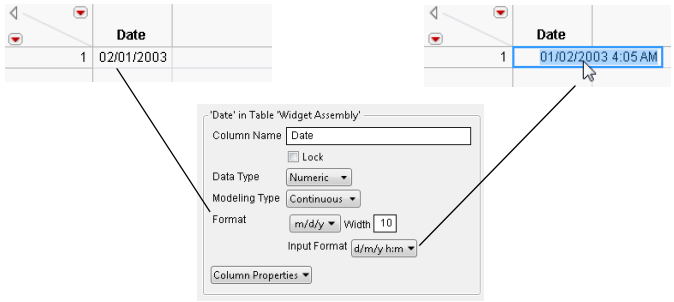
```
Format(3388649872,"ddMonyyyy h:m:s");
  "19May2011 11:37:52 AM"
Format(Today(),"ddMonyyyy h:m:s");
  "19May2011 11:37:52 AM"
```

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. The `Informat()` function defines the input format, and the `Format()` function defines the display format. For example,

```
New Table("Widget Assembly",
    Add Rows(1 ),
    New Column("Date",
        Numeric, "Continuous",
        Format("m/d/y" ),
        Informat("d/m/y h:m" ),
        Set Values([3126917100] )
    )
);
```

The `Format()` and `Informat()` 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 657 in the "Common Tasks" chapter for details.
- The date-separator character on your computer might differ from the forward slash (/) character shown in this book.
- 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 325 in the "Data Tables" chapter.

For descriptions of specific date-time functions, see the *JSL Syntax Reference*.

Table 6.4 Date-Time Formats

Type	Format argument	Example
Date only	`"m/d/y"`	`"01/02/1999"`
	`"mmddyyyy"`	`"01021999"`
	`"m/y"`	`"01/1999"`
	`"d/m/y"`	`"02/01/1999"`
	`"ddmmyyyy"`	`"02011999"`
	`"ddMonyyyy"`	`"02Jan1999"`
	`"Monddyyyy"`	`"Jan021999"`
	`"y/m/d"`	`"1999/01/02"`
	`"yyyymmdd"`	`"19990102"`

Table 6.4 Date-Time Formats *(Continued)*

Type	Format argument	Example
	"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"
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"

Table 6.4 Date-Time Formats *(Continued)*

Type	Format argument	Example
	"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.		
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"
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“

Currency

JMP displays numbers as currency using the `Format()` function, which uses the following syntax:

```
Format(x,"Currency", <"currency code">, <decimal>);
```

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"
```

If you do not specify the currency code, the currency symbol is based on the computer's operating system locale. 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"
```

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 6.5 lists the currencies supported in JMP.

Table 6.5 Currencies Supported in JMP

Code	Currency	Code	Currency	Code	Currency
AUD	Australian dollar	HKD	Hong Kong dollar	PHP	Philippine peso
BRL	Brazilian real	ILS	Israeli new shekel	PLN	Polish zloty
CAD	Canadian dollar	INR	Indian rupee	RUB	Russian ruble
CHF	Swiss franc	JPY	Japanese yen	SEK	Swedish krone
CNY	Chinese yuan	KRW	South Korean won	SGD	Singapore dollar
COP	Colombian peso	MXN	Mexican peso	THB	Thai baht
DKK	Danish krone	MYR	Malaysian ringgit	TWD	New Taiwan dollar
EUR	Euro	NOK	Norwegian krone	USD	US dollar

Table 6.5 Currencies Supported in JMP *(Continued)*

Code	Currency	Code	Currency	Code	Currency
GBP	British pound	NZD	New Zealand dollar	ZAR	South African rand

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 6.6.

These functions are listed in the *JSL Syntax Reference*.

Table 6.6 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.
`Hex To Char("hexstring", encoding)`	Returns a character string that corresponds to the hexadecimal code suppled as a quoted string. The default encoding supported for the hex code is utf-8. You can also specify one of these encodings: utf-16le, utf-16be, us-ascii, iso-8859-1, ascii-hex, shift-jis, and euc-jp.
`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 *string* of characters into a binary (blob). The default encoding supported for the hex code is utf-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.
`Blob To Char(blob)` `Blob To Char(blob, encoding)`	Converts binary data to a Unicode string. The default encoding supported for the hex code is utf-8. You can also specify one of these encodings: utf-8, utf-16le, utf-16be, us-ascii, iso-8859-1, and ascii-hex.

Table 6.6 Hexadecimal and BLOB Functions *(Continued)*

Syntax	Explanation
`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.

`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);
//result is "abcdef"

m=expr(mno);
```

```
c = m || "xyz";
show(c);
//result is an error message that mno is unresolved

m = expr(mno);
c = Name Expr(m) || "xyz";
show(c);
//result is "mnoxyz"

m=char(expr(mno));
c=m || "xyz";
show(c);
//result is "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 6.7 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*

Table 6.7 Munger behaviors for various types of arguments *(Continued)*

Find, *length*, and *replace* arguments	Example
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"`
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.

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()` 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, (<replacementString>, <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 also 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 argument in `Regex()` is a specification of the result string. In the previous example, the value defaults to `\0`, which means replace everything that was matched. The example could also be written as follows:

```
sentence = "I took the bus to work.";
Regex( sentence, "bus|car", "\0" );
   "bus"
```

`\0` is a backreference to everything that was matched by the regular expression.

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 157 for more information.

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, <NULL>, <MATCHCASE>);
```

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).

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"};

// Create the data table.
dt = New Table( "English 101",
   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 157.

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 6.8 describes the special characters and provides examples.

Table 6.8 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.

Table 6.8 Special Characters in Regular Expressions *(Continued)*

`$`	Matches the end of a string, not including the newline character. `apple$` 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".
`?`	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 6.9 describes the escaped characters supported in JMP. `\C`, `\G`, `\X`, and `\z` are not supported.

Table 6.9 Escaped Characters

`\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]`
`\l`	match a single lowercase letter `[a-z]`
`\L`	single character that is not lowercase `[^a-z]`
`\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 6.9 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, “there”, 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 ahead of 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 ahead of 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 157.

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( patMatch( "Take this bus.", p2 ), print("matches"),
  print("no match") ); // performing a 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 ),
  Show( vehicleType ),
  Print( "no match" )
); // do not use vehicleType in the ELSE because it is not set
```

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 = patLen(3)>>beginDelim + patArb()>?middlePart +
  expr(beginDelim);
testString = "SomeoneSawTheQuickBrownFoxJumpOverTheLazyDog'sBack";
rc = PatMatch( 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 `patAny("abc")`. The equivalent example for `patNotAny("abc")` is much harder. Similar to `patSpan` (above), `patBreak("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 = patSpan("0123456789") | "";

number = ( patAny("+-") | "" ) >? signPart +
         ( digits ) >? wholePart +
         ( "." + digits | "" ) >? fractionPart +
         ( patAny("eEdD") + ( patAny("+-") | "" ) + digits | "" ) >?
   exponentPart;

if( patMatch( "-123.456e-78", number ), show( signPart, wholePart,
   fractionPart, exponentPart ) );
```

Parsing Strings in Fixed Fields

Sometimes data is in fixed fields. The `patTab`, `patRTab`, `patLen`, `patPos`, and `patRPos` functions make it easy to split out the fields in a fixed field string. `PatTab` and `patRTab` 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 = patPos(10) + patTab(15);
```

`PatPos(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 `patTab(15)` matches text from the current position (10) forward to position 15. This pattern is equivalent to `patPos(10)+patLen(5)`. Another example:

```
p = patPos(0) + patRTab(0);
```

This example matches the entire string, from 0 characters from the start to 0 characters from the end. the `patRem()` function takes no argument and is shorthand for `patRTab(0)`; it means the remainder of the string. Pattern matching can also be anchored to the beginning of the string like this:

```
patMatch( "now is the time", patLen(15) + patRPos(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
patMatch( "babb", patPos(0) + p + patRPos(0) );
```

A recursive pattern refers to its current definition using `expr()`:

```
p = "<" + expr(p) + "*" + expr(p) + ">" | "x";
patMatch( "<<x*<x*x>>*x>", patPos(0) + p + patRPos(0) );
```

Remember, `expr()` is the procrastination operator; 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 `patMatch` 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 = patMatch( "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 );
   string = "abcABC"
   result = {"abcABC"}

result = Regex Match( string, Pat Regex( "[cba]+" ), NULL, MATCHCASE );
```

```
Show( string, result );
   string = "abcABC"
   result = {"abc"}
```

Chapter 7

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, other 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}
```

Assignments with Lists

Create a list to assign values to variables.

Examples

```
{a, b, c} = {1, 2, 3}; // assigns 1 to a, 2 to b, and 3 to c
{a, b, c}--; // decrements a, b, and c
{{a}, {b, c}}++; // increments a, b, and c by 1
mylist = {1, log(2), e()^pi(), height[40]}; // stores 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; // assigns 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;
```

Note the following:

- 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;
  ```

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.

Note the following:

- 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 details about matrix manipulation and a description of the equivalent `Loc()` command for matrices, see "Matrices" on page 173.

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

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 a pre-decrement operator. Use the SubtractTo() operator instead (-=).
||=	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.
	Eval List()	Eval List(*list*)	Returns a list of the evaluated expressions inside list. See "Evaluate Lists" on page 165.
	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.

Table 7.1 List Operators *(Continued)*

Operator and Function		Syntax	Explanation
	`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.
	`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.

Table 7.1 List Operators *(Continued)*

Operator and Function		Syntax	Explanation
	`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.
	`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 i^{th} item from the *list*. Subscripts can in turn be lists or matrices.

Table 7.1 List Operators *(Continued)*

Operator and Function	Syntax	Explanation
`Substitute()`	`Substitute(`*`list`*`,` *`pattExpr1`*`,` *`replExpr1`*`, ...)`	Returns a copy of a list or expression, replacing instances of each pattern expression with the corresponding replacement expression.
`Substitute Into()`	`Substitute Into(`*`list`*`,` *`pattExpr1`*`,` *`replExpr1`*`, ...)`	Changes a list or expression, replacing instances of each pattern expression with the corresponding replacement expression. **Note:** The *`list`* or expression must be a variable.

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 with `Concat()` or the `||` operator.

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}
```

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 188.

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 = [];
```

Specifying the number of rows and columns in an empty matrix is optional. JMP creates the matrix as necessary.

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 2nd and 3rd rows with the 2nd and 1st 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] = [];  // to delete the kth row
A[0, k] = [];  // to delete 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 `::` to create matrices of ranges.

```
T1=1::3; // creates the vector [1 2 3]
T2=4::6; // creates the vector [4 5 6]
T3=7::9; // creates the vector [7 8 9]
T=T1|/T2|/T3; // concatenates the vectors into a single matrix
T[1::3, 2::3]; // refers 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};
IsMatrix(A); // returns 1 for yes
IsMatrix(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
```

You can use the `Any()` or `All()` operators 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]; // yields [1]
a || {2}; // yields [2]
a || [3 4 5]; // yields [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]
```

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<<GetAsMatrix;

dt=Open("$SAMPLE_DATA/Big Class.jmp");
col=Column("Height");
A = col<<GetAsMatrix;
```

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<<GetAllColumnsAsMatrix;
```

Move Only Certain Columns

To get certain columns of a data table, use column list arguments (names or characters).

```
dt=current data table();
x=dt<<Get As Matrix({"height", "weight"});
```

or

```
x=dt<<Get As Matrix({height, weight});
```

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<<GetRowsWhere(Sex=="M");
```

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 `SetValues()` function copies values from a column vector into an existing data table column:

```
col<<SetValues(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<<SetValues(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);
```

Summarize Columns

There are several functions that return a row vector based on summary values for each column.

```
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 Sum(mymatrix) // returns the sum of each column
   [99 132]
V Std(mymatrix) // returns the standard deviations of each column
   [22 22]
```

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 as follows:

  ```
  colBox=Report(biv) [NumberColBox(13)];
  beta = colBox<<GetAsMatrix;
     [-127.145248610915,3.711354893859555]
  ```

Note the following:

- 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<<GetAsMatrix;
  ```

- 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<<GetAsMatrix;
  ```

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];
```

`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];
LocSorted(A,B);
   [3]

A = [10 20 40];
B = [35 5 45 20];
LocSorted(A,B);
   [2, 1, 3, 2]
```

Note the following:

- **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());
```

```
[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=vecdiag(
[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 `VecQuadratic()` 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 `VecDiag(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]
```

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 `DesignNom()` or `Design F()` function, which removes the last column and subtracts it from the others. Therefore, the elements of `DesignNom()` or `Design F()` matrices are 0, 1, and –1. And the `DesignNom()` or `Design F()` matrix has one less column than the vector or list has unique values. This operator makes full-rank versions of design matrices for effects.

```
x=[1, 2, 3, 2, 1];
DesignNom(x);
   [1 0,
   0 1,
   -1 -1,
   0 1,
   1 0]
```

`DesignNom()` is further demonstrated in the "ANOVA Example" on page 203.

To facilitate ordinal factor coding, use the `DesignOrd()` 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];
DesignOrd(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()`, `DesignNom()`, and `DesignOrd()` 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.

Note the following:

- 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 `DesignNom()` and `DesignOrd()`, 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]
DesignNom(20, [10 20 30]);
   [0 1]
DesignNom(30, [10 20 30]);
   [-1 -1]
DesignOrd(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.

```
HDirectProduct(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*AInv;
   [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);
   G Inverse(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) pivot diagonals 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, [...]); // where [...] indicates partition A
```

This produces the matrix result equivalent to the following:

$$\begin{bmatrix} A^{-1} & A^{-1}B \\ -CA^{-1} & D - CA^{-1}B \end{bmatrix}$$

Note the following:

- 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 "ANOVA Example" on page 203.

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 $n \times n$ matrices are defined recursively as a weighted sum of determinants for $(n - 1) \times (n - 1)$ 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 $Ax = \lambda x$ has a nonzero solution **x**. The λ's are called eigenvalues, and the corresponding **x** vectors are called eigenvectors. This is equivalent to solving $(A - \lambda I)\mathbf{x} = 0$. You can reconstruct **A** from eigenvalues and eigenvectors by a statement like the following:

```
newA=evecs*diag(evals)*evecs`;
```

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 193.)

  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, 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];
// conducts the Cholesky decomposition
exAchol = Cholesky( exS);

// adds 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];

// updates the Cholesky decomposition manually
exAnew = exS + exV * exC * exV`;
```

```
exAcholnew = Cholesky( exAnew);
```

Instead of manual updating, use `Chol Update()` to update the Cholesky decomposition, as follows:

```
// updates the Cholesky decomposition more efficiently
exAcholnew_test =
Chol Update( exAchol, exV, exC );

// results are the same as the manual process
Show( exAcholnew_test );
Show( exAcholnew );
```

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`.

Note the following:

- 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 `OrthoPoly()` 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.

```
OrthoPoly([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 199.

QR Decomposition

`QR()` factorization is useful for numerically stable matrix work. `QR()` returns a list of two matrices. The typical usage is as follows:

```
{Q, R} = QR(X);
```

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.

Note the following:

- 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 Operators

You can store your own operations in macros. See "Macros" on page 231 in the "Programming Methods" chapter. Similarly, you can create custom matrix operations. For example, you can make your own matrix operation called Mag() to find the magnitude of a vector, as follows:

```
mag=function({x},sqrt(x`*x));
```

Similarly, you could create an operation called Normalize to divide a vector by its magnitude, as follows:

```
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(nrow(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:

```
// open the data table
bigClass = open("$SAMPLE_DATA/Big Class.jmp");

// get data into matrices
x = (Column("Age")<<getValues) || (Column("Height")<<getValues);
x = j(nrow(x),1,1)||x;
y = Column("Weight")<<getValues;

// 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 = nrow(x)-ncol(x);  // degrees of freedom
mse = sse/dfe;          // mean square error, error variance estimate

// additional calculations on estimates
stdb = sqrt(vecDiag(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
newWindow("Big Class Regression",
  tableBox(
     StringColBox("Term",{"Intercept","Age","Height"}),
     NumberColBox("Estimate",  beta),
     NumberColBox("Std Error", stdb),
     NumberColBox("TRatio",    tratio),
     NumberColBox("Prob>|t|",  probt),
     NumberColBox("Lower95%",  betal95),
     NumberColBox("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:

```
designNom(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()`, as follows:

```
x = J(9,1,1) || designNom(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, as follows:

```
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:

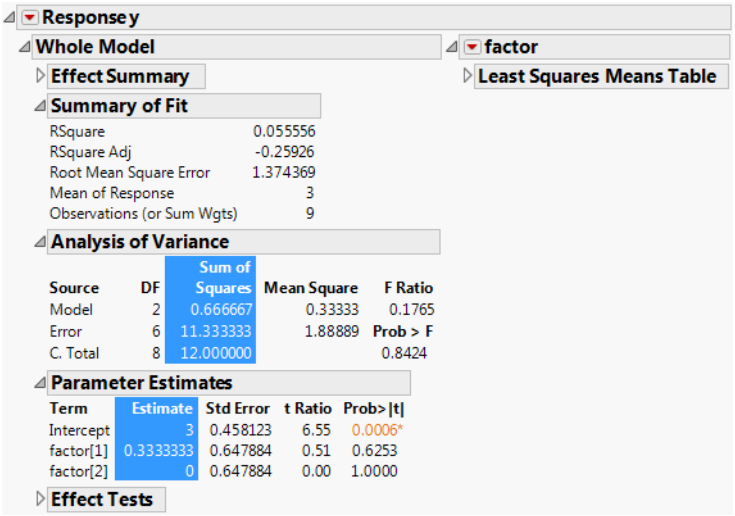
```
[0.111111111111111 0 0 3,
0 0.222222222222222 -0.11111111111111 0.333333333333333,
0 -0.11111111111111 0.222222222222222 0,
-3 -0.33333333333333 0 11.33333333333333]
```

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. See Figure 7.1.

Figure 7.1 ANOVA Report Within Fit Model



Construct the report in Figure 7.1 as follows:

1. Build a data table (described in the "Data Tables" chapter on page 275):

```
dt = New Table( "foo" );
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 "Launching Platforms" on page 382 in the "Scripting Platforms" chapter):

```
obj = Fit Model(
   Y( :y ),
   Effects( :factor ),
   Personality( "Standard Least Squares" ),
   Run Model(
      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 408 in the "Display Trees" chapter):

```
ranova = obj << report;
ranova[OutlineBox(6)] << Close(0);
ranova[OutlineBox(7)] << Close(1);
ranova[OutlineBox(9)] << Close(1);
ranova[OutlineBox(5), NumberColBox( 2 )] << select;
ranova[OutlineBox(6), NumberColBox( 1 )] << select;
```

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. Keys are placed in quotes. The value associated with a key can be a number, date, matrix, list, and so on.

Note: 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 instead is more efficient and faster.

Create Associative Arrays

To create an empty associative array, use the `Associative Array()` function or `[=>]`.

```
cary = Associative Array();
cary = [=>];
   [=> ]
```

Keys and values can be any JSL objects. Items can be added and changed with subscripting, as follows:

```
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);
   ["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.

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 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()`

Note the following:

- `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()`:

```
newcary = Insert(cary, "time zone", "Eastern");
show(cary, newcary);
   cary = Associative Array({
   {"high schools",{"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}
   });
   newcary = Associative Array({
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"time zone", "Eastern"},
   {"weather", "sunny"}
   });

Insert Into(cary, "county", "Wake");
show(cary);
   cary = Associative Array({
   {"county", "Wake"},
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}
   });
```

Note that `aa << 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`:

```
newcary = Remove(cary, "high schools");
show(cary, newcary);
```

```
   cary = Associative Array({
   {"county", "Wake"},
   {"high schools", {"Cary", "Green Hope", "Panther Creek"}},
   {"population", 116244},
   {"state", "NC"},
   {"weather", "sunny"}
   })
   newcary = ["county" => "Wake", "population" => 116244, "state" => "NC",
      "weather" => "sunny"];

Remove From(cary, "weather");
show(cary);
   cary = Associative Array({
   {"county", "Wake"},
   {"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

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 << getvalues;
dtbig = New Table( "BigBigClass",
   New Column( "name",
      character,
      setvalues( nms[J( 100000, 1, Random Integer( N Items( nms ) ) )] )
   )
);
Wait( 0 );
t1 = Tick Seconds();
Write(
   "\!N# names from BigBigClass = ",
   N Items( Associative Array( dtbig:name ) ),
   ", elpased time=",
   Tick Seconds() - t1
);
```

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);
   238
```

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 216.

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});
```

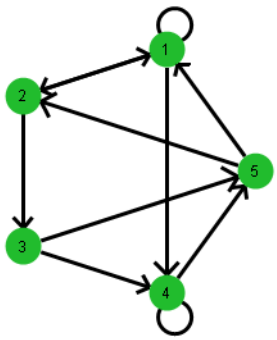
```
g[2] = Associative Array({1, 3});
g[3] = Associative Array({4, 5});
g[4] = Associative Array({4, 5});
g[5] = Associative Array({1, 2});
```

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 as follows:

- 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},
   Local( {chnode, tmp},
      Write( "Node: " || Char( node ) || ", " || Char( ref[node] << get
   contents ) || "\!N" );
      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;
      )
   );
```

Note the following:

- 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 this function works, try the following:

```
dfs( g, 2, J( N Items( g << get keys ), 1, 0 ) );
```

The output is as follows:

```
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 << getkeys,
   "\!N\!tset_z = ",
   set_z << getkeys,
   "\!N\!tset_w = ",
   set_w << getkeys
);
```

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 <<
   getkeys );
   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
   << getkeys );
   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 <<
   getkeys );
   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 << getkeys );
   Which of {ROBERT, JEFF, CHRIS, HARRY}, is not in Big Class = {"HARRY", "JEFF"}
```

Chapter 8

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

Stored expressions

An expression is something that can be evaluated. The first section of the chapter, "Rules for Name Resolution" on page 97 in the "JSL Building Blocks" chapter, discussed how JMP evaluates expressions. Now you must consider *when* JMP evaluates expressions.

JMP tends to evaluate things as soon as it possibly can, and it returns a result. If an expression is on the right side of an assignment, the result is what is assigned. Usually, that is what you want and expect, but sometimes you need to be able to delay evaluation.

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.)

```
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, not its result" on page 223.

```
show(nameExpr(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`.

```
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:

```
show(eval(a));
   Eval(a) = 21
```

You can do this to any level, for example:

```
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
```

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. Syntax coloring is also applied to the output.

The following script 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.
   ";
```

Store scripts in global variables

The main use of `Expr` is to be able to store scripts (such as macros) in global variables.

```
dist = expr(Distribution(Column(height)));
```

Now when you want to do the script, just mention the symbol:

```
dist;
```

You could even put it in a loop to do it many times:

```
for(i=0, i<10, i=i+1, dist);
```

You can 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, not its result

What if you wanted the symbolic value of a global (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, but if the argument is a name, it looks up the name's expression and uses that, unevaluated.

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()/nrow() ), 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 `i`th 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 `i`th argument
- `Arg(`*`expression`*`,i)` evaluates *`expression`* and finds the `i`th argument
- `Arg(Expr(`*`expression`*`),i)` finds the `i`th 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 lots of substitutions

`Eval Insert` is for the situation where you want to make a lot of substitutions, by evaluating expressions inside a character string. [In Perl, this is called *interpolation*.]

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 = EvalInsert("abc^xstring^ghi");  // results in "abcdefghi";

// in-place evaluation
r = "abc^xstring^ghi";
EvalInsertInto(r); // r now has "abcdefghi";

// with specified delimiter
```

```
r = EvalInsert("abc%xstring%ghi","%");  // results in "abcdefghi";

// with different start and end delimiters
r = EvalInsert("abc[xstring]ghi","[","]");  // results in "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.

```
EvalInsert( "^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 = evallist(x);   //result in 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 229 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 global 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);
```

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);
  // results in 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) );
  // results in 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 operators with *x*. Assume that a data table contains a column named X3, and *x* and *i* have been assigned:

```
// assumes a data table with column named X3
x =Expr( Distribution(column( Expr("X"||char(i)) )) );
i = 3;
```

Table 8.3 Compare all the operators 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 Compare all the operators 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. `Eval(` `EvalExpr(` `Parse(` `Char(` `NameExpr(x)))));` *`Distribution(Column(Expr("X" || Char(i))))`*

Macros

Stored expressions can serve as a macro feature. You can store a generalized action as an expression in a global, and then call that global 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 (the script itself, not its evaluation at the moment) with `Expr` delays its evaluation until the global is actually called. Any variables, data points, or expressions included in that expression are evaluated on the fly when the expression is evaluated. See "Stored expressions" on page 221, for detailed rules for storing expressions and later evaluating them.

Manipulating lists

The following operators manipulate lists. They can also be used to manipulate expressions, as shown in the next section, "Manipulating expressions" on page 233. A summary of commands with explanations is in Table 8.4 on page 235.

Most of the function have two variants, one that produces a new value, and one that works in-place directly on its arguments. Here are some example pairs:

```
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

onetwo=Insert({1},2);  // onetwo is {1,2}
InsertInto(A,{1,2},4); // puts 1,2 before the current 4th item
```

Note: If position is omitted in the `Insert Into` command, items are placed at the end of the list.

```
a=Shift({1,2,3,4},1); // stores the list {2,3,4,1} in a
Shift Into(a,-1); // a is now {1,2,3,4}

b=Reverse(a); // b is now {4,3,2,1}
Reverse Into(a); // a is now {4,3,2,1}, also

s=Sort List({1,4,2,5,-7.2,pi(),-11,cat, apple, cake});
  // s is now sorted list
```

```
c={5,pie,2,pi(),-2};
Sort List Into(c); // c is now {-2,2,5,Pi(),pie}
```

In-place operators

In-place operators 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 operator must be an L-value. An *L-value* is an entity such as a global 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 operators

For the *not-in-place operators,* you must either state the list directly or else quote a name that evaluates to a list. Such operators 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 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); // produces {23,b,c}
Substitute(expr(sine(x)),expr(x),expr(y)); // produces 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,nameexpr(canine),nameexpr(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,
   nameexpr(quick),nameexpr(fast),
   nameexpr(brown),nameexpr(black),
   nameexpr(fox),nameexpr(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 operators 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); // results in the expression Log(3)^3/3
```

As with lists, remember that the first argument for in-place operators must be an L-value. An L-value is an entity such as a global variable whose value can be set. In-place operators 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);
insertinto(polynomial,expr(d*x^3),1);
show(polynomial);
  polynomial = d * x ^ 3 + a * x ^ 2 + b * x + c
```

For the not-in-place operators, you must either state the expression directly or else quote a name that evaluates to an expression using `NameExpr`. Such operators 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(nameexpr(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 232. 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(NameExpr(a));
  NameExpr(a) = Distribution(Column(x), normal quantile plot)
```

`Substitute` evaluates all its arguments, so they must be quoted correctly:

```
b = substitute(NameExpr(a),expr(x),expr(weight)); show(NameExpr(b));
  NameExpr(b) = Distribution(Column(weight), normal quantile plot)
```

`SubstituteInto` needs an L-value, so the first argument is not quoted:

```
SubstituteInto(a,expr(x),expr(weight)); show(NameExpr(a));
  NameExpr(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(nameexpr(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))};
// Next, plug in the coefficients:
substitute into(x,expr(a),4, expr(b),0, expr(c),-9);
show(x);           //see the result of substitution
show(evalexpr(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))}
  EvalExpr(x) = {1.5,-1.5}
```

The operators for manipulating lists and expressions are discussed in the previous section, "Manipulating lists" on page 231, 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.

Table 8.4 Functions for manipulating lists or expressions *(Continued)*

Function	Syntax	Explanation
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.
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.

Table 8.4 Functions for manipulating lists or expressions *(Continued)*

Function	Syntax	Explanation
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.
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 238.
- Scopes that are pre-defined by JMP. See "Scoped Names" on page 240.
- Namespaces that you can create for your scripts. See "Namespaces" on page 244.

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. We 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.

`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.

If you have simple scripting needs, `Names Default To Here(1)` might be sufficient.

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 240 for details 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.

```
// Script 1
a = 1;
show(a);

// Script 2
a = 3;
show(a);
```

1. Run Script 1. The result is as follows:

```
a = 1
```

2. Run Script 2. The result is as follows:

```
a = 3
```

3. Run only the `show(a);` line in Script 1. The result is as follows:

```
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. The result is as follows:

```
a = 1
```

6. Run Script 2. The result is as follows:

```
a = 3
```

7. Run only the `show(a);` line in Script 1. The result is as follows:

```
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 244.

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.

Table 8.5 Predefined Scopes *(Continued)*

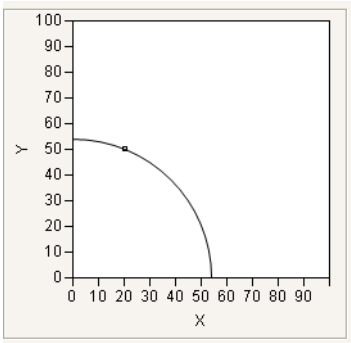
Scope	Description
`Builtin`	JMP built-in functions. For example, `Builtin:Sqrt()`. These names are shared throughout the JMP environment. If you over-ride a JSL function with a custom function, you can still access the built-in JSL function by using this scope.
`Local`	Nearest local scope. Can be nested within the user-defined functions, `Local` and `Parameter`.
`Local Here`	Provides a namespace block inside `Names Default to Here(1).` `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.)
`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.

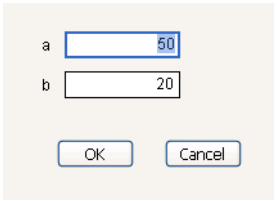
This script produces two windows and uses two different scopes.

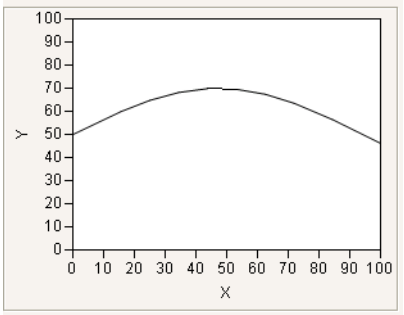
The Launcher window asks the user for two values. Those two values are passed to the Output window, which uses them to graph a function. The Launcher window scopes `aBox` and `bBox` to the window: essentially, those two variables (pointers to Number Edit Boxes) exist only in the Launcher window and are 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 244 for details 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`.

Table 8.6 defines the messages that are supported by user-defined namespace references.

Table 8.6 Namespace Messages

Namespace Message	Description
`ns << Contains( "var_name" );`	Returns 1 or 0, depending on whether *var_name* exists within the namespace.
`ns << Delete;`	Removes this namespace from the internal global list. To delete variables in the namespace, use `<<Remove`. See the entry for `<<Remove` in this table.
`ns << First;`	Returns a quoted string that contains the first variable name used within the namespace.

Table 8.6 Namespace Messages *(Continued)*

Namespace Message	Description
`ns << Get Contents;`	Returns a list of key-value pairs, which are each enclosed in a list. Each key is a quoted string that contains a variable name, and each value is the unevaluated expression that the variable contains.
`ns << Get Keys;`	Returns a list of variable names.
`ns << Get Name;`	Returns the name of this namespace.
`ns << Get Value( "var_name" );`	Returns the unevaluated expression that *var_name* contains in this namespace.
`ns << Get Values;`	Returns a list of unevaluated expressions that each variable in this namespace contains.
`ns << Get Values( { "var_name1", "var_name2", ... } );`	Returns a list of unevaluated expressions that each variable in this namespace specified in the list argument contains. If a requested variable name is not found, an error is returned.
`ns << Insert( "var_name", expr );`	Inserts into this namespace a variable named *var_name* that holds the expression *expr*.
`ns << Lock;` `ns << Unlock;`	Locks all variables in the namespace and prevents variables from being added or removed. `<<Unlock` unlocks all of the namespace's variables.
`ns << Lock( <"var_name", ...> );` `ns << Unlock( <"var_name", ...> );`	Locks the specified variables in this namespace. If no variables are specified, all variables are either locked or unlocked.
`n = ns << N Items;`	Returns the number of variables contained in this namespace.
`next k = ns << Next( "var_name" );`	Returns the name of the variable that follows the specified variable.
`ns << Remove( "var_name", ...);`	Removes the specified variable or list of variables.

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 namespaces is an anonymous namespace 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`.

See "Includes" on page 258 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 2-element lists to represent complex numbers, and then locks the namespace.

```
If( !Namespace Exists( "complex" ),
  New Namespace( "complex" );

  complex:make = Function( {a, b}, Eval List( {a, b} ) );
  complex:add = Function( {a, b}, a + b );
  complex:subtract = Function( {a, b}, a - b );
  complex:multiply = Function( {a, b}, Eval List( {a[1] :* b[1] - a[2] :*
  b[2], a[1] :* b[2] + a[2] :* b[1]} ) );
  complex:divide = 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:char = 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:make( 3, 4 );
  {3, 4}

c2 = complex:make( 5, 6 );
  {5, 6}

cm1 = complex:make( [1, 2, 3], [4, 5, 6] );
  {[1, 2, 3], [4, 5, 6]}

cadd = complex:Add( c1, c2 );
  {8, 10}

csum = complex:Subtract( c1, c2 );
  {-2, -2}

cmul = complex:Multiply( c1, c2 );
  {-9, 38}

cdiv = complex:Divide( c1, c2 );
  {14.6065573770492, 19.7049180327869}

show( complex:char( c1 ) );
  complex:char(c1) = "3+4i";
```

Referencing Namespaces and Scopes

There are a number of factors in resolving a named variable reference. Table 8.7 describes the named variable references that are resolved for specific situations.

Table 8.7 Namespace References[a]

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[b] • `Here` namespace • current data table If the *Names Default To Here* mode is off, JMP looks for the variable in these locations: • `Local` namespace[b] • `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[b] or in the `Here` namespace. • If the *Names Default To Here* mode is off, then JMP creates the variable in the `Local` namespace[b] 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 8.7 Namespace References[a] *(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 252.	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.

a. These forms existed in JMP 8. In JMP 9 and later, a, :a, and ::a have the same meaning with the *Names Default To Here* mode turned off.
b. 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.

Example of Local:a

Sample Script	Log Output
`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;* ****Error={"Name Unresolved: d1l1"(1, 2,* *"d1l1", d1l1 /*###*/)}* *// Local* *d2l1 = 56;* *l2l2 = 78;* *// 2 Local* *// Local* *fa1 = 2;* *fa12 = 5;* *fa2 = 3;* *fl1 = 99;* *// 4 Local* *// Local* *d1l1 = 12;* *// 1 Local* *// Global* *exception_msg = {"Name Unresolved:* *d1l1"(1, 2, "d1l1", d1l1/*###*/)};* *// 1 Global* *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 details about variable name resolution, see the "Rules for Name Resolution" on page 97 in the "JSL Building Blocks" chapter.

Rules for Resolving Variable References

To resolve variable references, JMP uses the following rules (in the order indicated):

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, look it up as a function 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.
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.

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 complex scripts.

Throwing and Catching 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.

`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.

`Throw` does not require an argument but has two types of optional arguments. If you include a character-valued expression as an argument, throwing stores that string in a global named *exception_msg*; this is illustrated in the first example below.

Examples

For example, you can use `Try` and `Throw` to escape from deep inside `For` loops.

```
a = [1 2 3 , 4 5 . , 7 8 9];
b = a;
nr = nrow(a);
nc = ncol(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(isMissing(za),throw("Missing a"));
         zb = b[j,i]; if(isMissing(zb),throw("Missing b"));
         sum += za*zb;
      )
```

```
    ),
    show(i,j,exception_msg); throw();
);
    i = 2;
    j = 3;
    exception_msg = "Missing a";
```

You can also use Try and Throw to catch an exception that JMP itself throws:

```
try(
    dt=open("My dataset.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();
);
```

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 (nrow(CurrentDataTable())==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);  // result in a is 2
b = myRoot(-1); // result in b is 0
```

Functions are stored in globals, 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(
  NameExpr(MakeFunction),
  expr(x), expr(y)
);
show(d);
makeFunction;
```

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 as follows.

```
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);
```

The use of `Default Local` localizes all the names that:

- 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);
```

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);
```

In both cases, the variable `temp` is not a global, or, if it is already a global, remains untouched by evaluating the functions.

Note: If you use an expression initially as local, then use it as a global, JSL changes the context. However, an expression used globally stays resolved globally regardless of its future use.

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. The user should enumerate each and every variable they wish 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 237 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.

Loading and Saving 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.

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 are some examples:

```
New Column( "Mean of height by sex", Numeric, Formula( Col Mean( :height, :sex
  ) ) );

New Column( "Minimum of height by sex and age", Numeric, Formula( Col Minimum(
  :height, :sex, :age ) ) );

Distribution( Continuous Distribution( Column( :height ) ), By( :sex ) );

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 ) )
  )
);
```

Organize Files into Projects

JMP projects are fully scriptable. The following script creates a new project, adds groups and files to it, and retrieves the project's name:

```
expj = New Project( "My Project" );
exg1 = expj << Add Group( "Data" );
exg2 = expj << Add Group( "Reports" );
exdt = Open( "$SAMPLE_DATA/Big Class.jmp" );
exg1 << Add Window( exdt );
exrp = Bivariate( X( height ), Y( weight ) );
exg2 << Add Window( exrp );
Close( exdt, NoSave );
expj << getname();
```

To open a project that already has been saved, use the `Open Project` function. For example,

```
prj = Open Project("filepath");
```

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<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;=QQTCEG??U<PUXLV?TRBO?J>QGWTJCFJA@BNHWLVORNNGQYPIKL<IM<>JX>@G?LJ>=;RBODH@PTKK@SIUE;IJOR<TUTRMTGSYRSVGOR<XK<F=IWQYE=LVZFP;AUHA?YJLL;EIT?ZJZC;*
```

7. Save the encrypted script.

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.

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 8.3 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;;;*"
)
```

Additional Numeric Operators

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 the chapter "Matrices" on page 173 in the "Data Structures" chapter, for details.

Derivatives

JSL has three internal operators (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, we first show the mathematical notation and then 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}
```

`NumDeriv` takes the first numeric derivative of an operator or 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. `NumDeriv2` 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

```
x=3;
n=NumDeriv(3*x^2);
```

The *correct* way is to make *x* an argument in the function.

```
x=3;
f=function({x}, 3*x^2);
n=NumDeriv(f(x), 1);
```

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=numderiv(x^2);  // or equivalently: y = numDeriv(3^2);
   6.0000099999513
```

And here are a few more examples:

```
x = numderiv(sqrt(7));
y = numDeriv(Normal Distribution(1));
z = Num deriv2(normal distribution(1));
```

Table 8.8 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

$y = \text{sqrt}(\log(x))$

$y^2 = \log(x)$

$\exp(y^2)=x$

`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 = simplifyExpr(
  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.

The form of the call is

```
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
<< maxIter( 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.

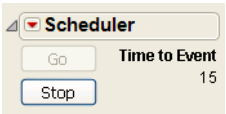
```
xx = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900,
  1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990];
yy = [3.929, 5.308, 7.239, 9.638, 12.866, 17.069, 23.191, 31.443, 39.818,
  50.155, 62.947, 75.994, 91.972, 105.71, 122.775, 131.669, 151.325, 179.323,
  203.211, 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 );
```

Scheduling Actions

A `Schedule` function lets you set up a script to be executed some number of seconds later.

```
schedule(15, print("hello"));
```

Figure 8.4 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 8.9 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.

JMP's scripting language has methods for constructing dialog boxes to ask for data column choices and other types of information. See "Modal Windows" on page 461 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, a colon `:`, 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 colon.

```
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.

```
write("Here is a message.");
   Here is a message.

myText="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 87 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 Macintosh, 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:

```
// Time Announcer
script = expr(
        tod = mod(today(),indays(1));
        hr  = floor(tod/inHours(1));
        min = floor(mod(tod,inHours(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.", "C:\myJMPData\Big Class.jmp");
```

Notes:

- On Macintosh, `Mail()` works on Mountain Lion and Mavericks. 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.
- On Windows, the bitness of JMP and the e-mail client must match. For example, `Mail()` does not work in JMP 32-bit with Microsoft Outlook 64-bit.

Chapter 9

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 this book, *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

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 "Working with the Log" on page 66 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, refer to the Scripting Index:

1. Select **Help > Scripting Index**.
2. Set the menu to show **All Categories**.
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 369.

```
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. This section covers the basic actions that you can perform on a data table, such as naming, saving, resizing, and so on.

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"); // just open the data table
  ```

- To open a data table and retain a reference to it:

  ```
  dt = Open("$SAMPLE_DATA/Big Class.jmp"); // open and retain 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 (Macintosh).

```
Open("../My Data/Repairs.jmp"); // relative path on Windows and Macintosh
Open("::My Data:Repairs.jmp"); // relative path on Macintosh
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 details about path variables, see "Path Variables" on page 126 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( "$SAMPLE_DATA/Big Class.jmp" );
obj = Bivariate( Y( :height ), X( :age ), Fit Line );
Close( dt );
If( Not( Is Scriptable( dt ) ),
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" ),
);
obj = Oneway( Y( :height ), X( :age ), Means( 1 ), Mean Diamonds( 1 ) );
```

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.

Invisible | Private

Invisible is an optional keyword that hides the new table from view but shows it in the Window List pane. Private is an optional keyword that hides the new table from view and from the Window List pane.

Making a table private can prevent memory problems when a script opens and closes many tables. Private data tables have no physical window, so less memory is required than with invisible tables.

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 three-letter 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("My Data.xlsx"); // Microsoft Excel file
Open("My Data.txt"); // text file
Open("$SAMPLE_IMPORT_DATA/Carpoll.xpt"); // SAS transport file
```

On Macintosh, JMP relies on the Macintosh 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 Macintosh.

- 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("My Data.txt", text);
  ```

Note: On Macintosh, if you do not specify the text file type, the text file opens in the Script Editor.

Additional supported formats include .csv, .jsl, and .jrn.

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 settings are optional.

The following `Open()` options are available:

```
CharSet("option")
```

```
    // "Best Guess", "utf-8", "utf-16", "us-ascii", "windows-1252",
    "x-max-roman", "x-mac-japanese", "shift-jis", "euc-jp", "utf-16be",
    "gb2312"
Number of Columns(Number)
Columns(colName=colType(colWidth),... )
    // colType is Character|Numeric
    // colWidth is an integer specifying the width of the column
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:

```
Open("$SAMPLE_IMPORT_DATA\EOF_space.txt", Labels(0), End of Field(Space));
```

The following sections describe each argument in more detail. For more detailed information about import options, refer to the *JSL Syntax Reference*.

Number of Columns

Specifies the 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 325.

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. The choices are as follows:

- CR for carriage returns (typical for text files created on Macintosh OS 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. Note the following:

- 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.

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 the *Using JMP* book for details about Microsoft Excel support.

In the JMP preferences, settings in the General group can help determine how worksheets are imported:

- **Excel Open Method** specifies how an 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 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 spreadsheet 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 below in your JSL scripts.

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 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 592 in the "Extending JMP" chapter for details.

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 Macintosh
```

Import a Web Page

A web page can include data in tabular format. Import the table as a JMP data table as follows:

```
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 headers defined in <th> HTML tags. These headers are converted to column headings in the data table.

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 file path can be absolute or relative. The following examples show how to open files from a shared computer named *Data*. If you plan to share the script, it's safer to use a relative path to the computer, not a path to the mapped drive.

```
Open("\\Data\Repairs.jmp");
Open("\\Data\My Data.txt");
```

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. For details, see *Essential Graphing*.

Import a Password-Protected Microsoft Excel 2007 File

JMP does not support importing password-protected .xlsx files. Import password-protected Excel 2007 .xls files by including the `Password` argument.

```
Open("Housing.xls", Password( "helloworld" ));
```

Note: Password-protected Excel version 2010 and 2013 .xlsx files are not supported even if you save the file as .xls.

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 587 in the "Extending JMP" chapter for more information.

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, as follows:

  ```
  dt1 = Open("$SAMPLE_DATA/Big Class.jmp");
  ```

- When you create a new table, as follows:

  ```
  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); //makes 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 "Sending Script Commands to a Live Analysis" on page 378 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.xls"); // save as a Microsoft Excel file
```

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 (Macintosh). For details about setting the default directory, see "Relative Paths" on page 129 in the "Types of Data" chapter.

On Windows, saving with a .txt extension exports according to the Text Export preferences. On Macintosh, add Text as a second argument to the Save function, as follows:

```
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 as follows:

```
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", newColumn("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 Macintosh, 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" );
```

Making a table private can prevent memory problems when a script opens and closes many tables. Private data tables have no physical window, so less memory is required than with invisible tables.

As with invisible tables, analyses that are run on a private data table are linked to the table.

To avoid losing a private data table, you must assign it a reference as shown in the preceding example. Otherwise, JMP immediately removes the private data table from memory. Additional uses of the table later in the script generate errors.

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 `<< Maximize Display` forces the data table to re-measure all of its columns and zoom to the best-sized window.

```
Current Data Table() << Maximize 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);
```

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);
```

Tip: `Close All()` also works with journals and layouts. Specify "Journals" or "Layouts" in the argument.

To save a copy with a new name and then close the original version:

```
Close(dt, Save("c:/My File.jmp"));
```

Set and Get a Data Table

When using a `DataBrowserBox` created using `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, use `N Table` to get a list of references to each one:

```
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 listbox (dialog) so that the user can choose and perform tables operations. Or, you can write the names out to a file.

Create Journals and Layouts

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.

Layouts enable you to combine several reports or rearrange report elements before saving them in one layout file.

The basic syntax for journals and layouts appears as follows:

```
Current Data Table() << Journal;
Current Data Table() << Layout;
```

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 and layout:

```
New Window( <<  Journal);
New Window( <<  Layout);
```

The following examples create an empty journal and layout named “Sales”:

```
New Window("Sales", << Journal);
New Window("Sales", << Layout);
```

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.

The four options for arguments are as follows:

- 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"));
```

Advanced Data Table Scripting

This section covers more advanced actions that you can perform on a data table, which includes collecting summary statistics, subsetting, sorting, concatenating, and so on.

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 )
);
```

Named arguments include the following: `Count`, `Sum`, `Mean`, `Min`, `Max`, `StdDev`, `First`, `Corr`, and `Quantile`. Each argument takes a data column argument.

Note the following:

- 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.

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:

```
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 `TableBox`. For details, see "Build Your Own Displays from Scratch" on page 452 in the "Display Trees" chapter.

```
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 9.1 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, as follows:

```
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 9.2 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, as follows:

```
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:

```
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 296 for details.

```
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:

```
summDt = dt << Summary( Group( :age ), Mean( :height ),
  Link to original data table( 0 )
);
```

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 specified, it uses all columns. And if `Subset` has no arguments, the Subset window appears.

```
dt << Subset(
  Columns(columns),
  Rows(row matrix),
  Linked,
  Output Table Name("name"),
  Copy Formula (1 or 0),
  Sampling rate (n),
  Suppress Formula Evaluation(1 or 0));
```

Note: For more arguments, see the Scripting Index.

For example, using Big Class.jmp, to select the columns for all rows in which the age is 12:

```
For Each Row(Selected(Rowstate())=(age==12));
subdt = dt << Subset(output table name("subset"));
```

To select three columns and all rows:

```
subDt1 = dt << Subset (Columns(Name,Age,Height), Output Table Name("Big Class
  NAH"));
```

To select specified rows of two columns, linking:

```
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:

```
dt << Select Where(age==12);
dt << Subset(( Selected Rows ), Output Table Name("subset"));
```

Subset Data Using the 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.

The basic syntax appears as follows:

```
dt << Data Filter( <local>, <invisible>, <Add Filter (...)>, <Mode>, <Show
  Window( 0 | 1 ), <No Outline Box( 0 | 1 )> );
```

Options for `Data Filter` include the following:

```
Add Filter, Animation, Auto Clear, Clear, Close, Columns, Conditional, Data
  Table Window, Delete, Delete All, Display, Get Filtered Rows, Location,
  Match, Mode, Report, Save and Restore Current Row States, Set Select, Set
  Show, Set Include, Show Column Selector, Show Subset, Use Floating Window
```

Note: For more arguments, see the Scripting Index.

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(bool), Show(bool), Include(bool)`. 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. The basic syntax appears as follows:

```
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, as follows:

```
df = dt << Data Filter(
  Mode( Show( 1 ) ),
  Add Filter(
     columns( :age, :sex, :height ),
     Where( :age == {13, 14, 15} ),
     Where( :sex == "M" ),
     Where( :height >= 50 & :height <= 65 )
```

```
    )
);
```

This script selects 13, 14, and 15-year-old males with a height greater than or equal to 50 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:

```
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"} ) ) ),
      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 the *Using JMP* book for details on the options for the Multiple Response property.

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 as follows:

```
obj  = ( Current Data Table() << Data Filter( Show Window(0) ) );
```

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.

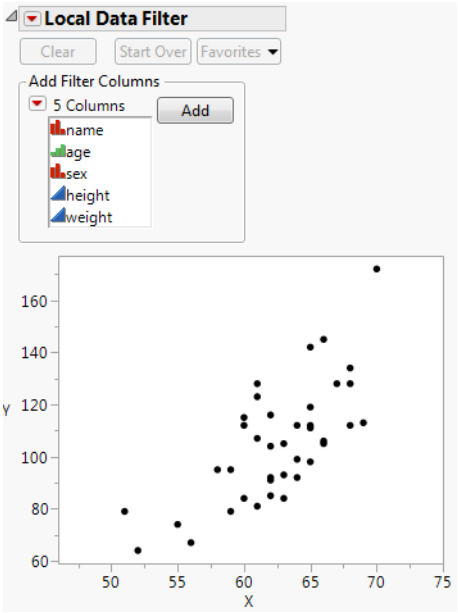
The following example creates a local data filter for a Graph Box. See Figure 9.4 for the output.

```
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: To experiment with this script, open the Local Data Filter for Custom Graph.jsl sample script.

Figure 9.3 Local Data Filter and Graph



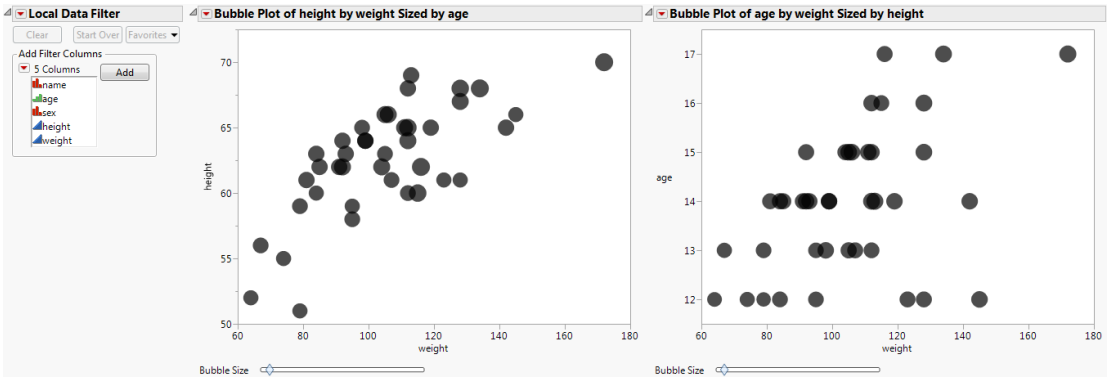
A context box can also contain several graphs with one local filter. The filtering then applies to both graphs. The following script creates two bubble plots and one data filter in one context box. See Figure 9.4 for the output.

```
Data Filter Context Box(
  H List Box(
     dt << Data Filter( Local ),
     Platform(
```

```
            Current Data Table(),
            Bubble Plot( X( :weight ), Y( :height ), Sizes( :age ) )
        ),
        Platform(
            Current Data Table(),
            Bubble Plot( X( :weight ), Y( :age ), Sizes( :height ) )
        )
    )
);
```

Tip: To experiment with this script, open the Local Data Filter Shared.jsl sample script.

Figure 9.4 Local Filter with Two Bubble Plots



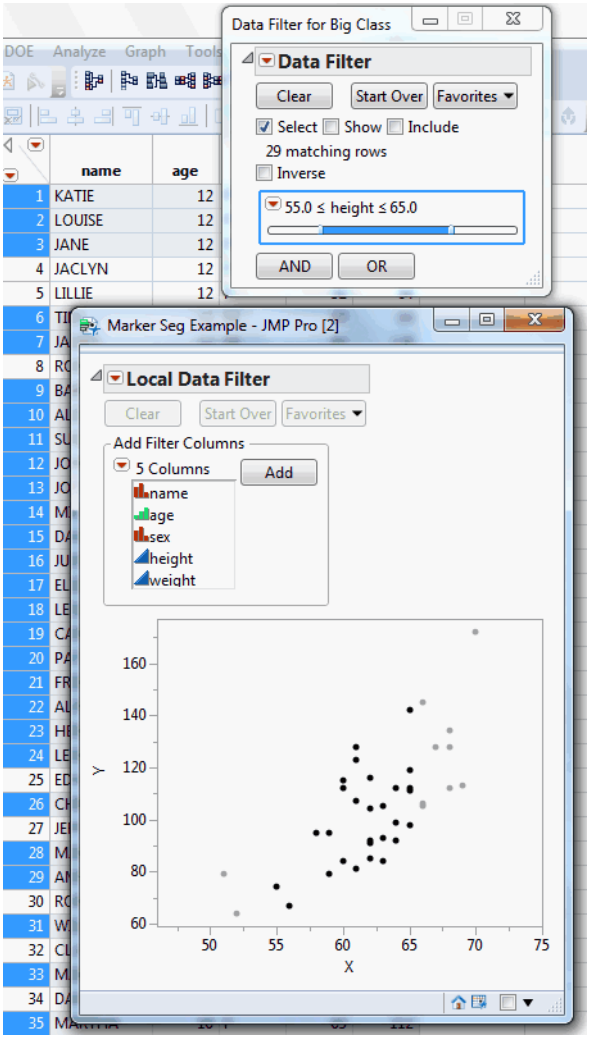
Data filters can be hierarchical. One script might generate a data filter and a local data filter. The outer data filter for the data table determines which data is available for the local data filter. To illustrate this point, the following script produces both types of data filters as shown in Figure 9.5:

```
dt << Data Filter(
   Add Filter( Columns( :height ), Where( :height >= 55 & :height <= 65 ) )
);
// local data filter
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 9.5 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) );
```

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. For details, see "Sort a Column's Values in Lexicographic Order" on page 212 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
   Keep (columns), //the columns to keep in the data table
   Drop (columns), //the columns to drop in the data table
   Output Table ("name"), // name for the new data table
   Columns(columns)); // specify which columns to include in the stacked table
```

For example, where *dt* is a reference to Big Class:

```
dt = Open("$SAMPLE_DATA/Big Class.jmp");
dt << Stack(
    columns( :weight, :height ),
    Source Label Column( "ID" ),
    Stacked Data Column( "Y" ),
    Name( "Non-stacked columns" )(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.

Split Values in a Stacked Data Table

Split breaks a stacked column into several columns.

```
dt << Split(
  Split(columns),  // the column to split
  Split by(columns) // the columns to split by
  Group(columns),  // (optional) column to identify rows uniquely
  ColID(column),   // the grouping variable on which to split
  Remaining Columns( Keep All | Drop All | Drop(columns) | Keep(columns)),
  Sort by Value Order //uses the Value Ordering column property to sort the
  output
  Output Table ("name"));
```

The optional Remaining Columns argument specifies which of the other columns from the source data table (those not specified for Split, Group, or ColID) to include in the new data table. The default is to Keep All, or you can explicitly list columns to Keep or Drop.

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:

```
splitDt = stackedDt << Split(
  split(y),
  ColID(ID),
  output table name("Split"));
```

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 << Concatenate(DataTableReferences, 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 first table
  With(dataTable), // the other data table
  Select(columns),     // optional column selection
  Select With(columns), // optional column selection
  By Row Number, // join type; alternatives are Cartesian Join or
                 // By Matching Columns(col1==col2,col),
                // Update first table with data from second table,
                // Merge same name columns,
  Copy formula(0), // on by default; 0 turns it off
  Suppress formula evaluation(0), // on by default; 0 turns it off
  // options for each table:
     Drop Multiples(Boolean, Boolean),
     Include NonMatches(Boolean, Boolean),
Preserve Main Table Order(); // maintains the order of the original data
```

```
Output Table Name("name")); // the resulting table
```

To try this, first break Big Class.jmp into two parts:

```
part1=dt << Subset(Columns(Name, Age, Height), Output Table Name("NAH_Big
  Class"));
part2=dt << Subset(Columns(Name, Sex, Weight), Output Table Name("NSW_Big
  Class"));
```

To make it a realistic experiment, rearrange the rows in part 2:

```
sortedPart2=part2 << sort(by(name), Output Table Name("SortedNSW_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.

```
joinDt = 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.

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, as follows:

```
NewHt=dt << Subset(Columns(Name, Height), Output Table Name("hts"));
```

Next, add 0-6 inches to each student's height:

```
diff = random uniform(6,0);
For Each Row(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
        Analysis Columns( :height, :weight ),
          // use the height and weight columns for the analysis
```

```
        Statistics( Std Dev, Mean ) // show the standard deviation and mean
)));
```

Note: You can also change the width of columns and perform additional operations within Tabulate. For details, see the JMP Scripting Index.

Find Missing Data Patterns

If your data table contains missing data, you might want to see whether there is a pattern that the missing data creates.

```
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 the following function:

```
mtx = (obj << Get Difference Summary matrix);
```

The resulting matrix appears in the log window, for example:

```
[-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).

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.

The basic syntax is as follows:

```
dt << Subscribe( "name"(<"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.

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. Unsubscribe as follows:

```
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 ));
```

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. JMP asks the user to confirm when trying to close the data table.

```
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 );
Current Data Table() << 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 183 in the "Data Structures" chapter.

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.

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);     // assigns a reference to the second column
```

This book uses *col* to represent data column references. To see the messages that you can send to data column objects, refer to the Scripting Index (**Data Tables > Column Scripting**). Alternatively, you can use the `Show Properties` command, as follows:

```
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, as follows:

```
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
currentDataTable()<<Select Where(col[]<14);  // in a WHERE clause
```

Create Columns

To add a new column to a data table, send a `New Column` message to a data table reference. 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 = Current Data Table();
dt << New Column("wafer");
```

or

```
a = "wafer";
dt << New Column(a);
```

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).

Unless otherwise specified, columns are numeric, continuous, and 10 characters wide.

- Data type (numeric, character, or row state)
- Analysis type (continuous, nominal, or ordinal)
- Column width (only for numeric columns)
- Numeric format

The following example creates a new column whose data type is numeric, analysis 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 analysis type.

```
dt << New Column("Last Name", Character);
```

You can also add formulas and other script messages appropriate to the column.

```
dt << New Column("Ratio", Numeric, "Continuous", Formula(Height/Weight));
dt << New Column("myMarkers", Row State, Set Formula(Marker State(age-12)));
```

If you plan to work with specific columns later (for example, grouping or changing data types), create a column reference, as follows:

```
myCol = dt << New Column("Part Number");
```

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 the current data table. Three values are added: Smith, Jones, and Anderson.

```
dt = Current Data Table();
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 the current 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 = Current Data Table();
dt << New Column("Row Number", Numeric, Values(1::NRow()), Format("Best",5));
```

To add a column of random numbers, insert a formula along with the random function.

```
dt = Current Data Table();
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.

```
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("logHt"); // command is sent to the data table reference
New Column("logHt"); // column is added to the current data table
```

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.

```
dt = Current Data Table();
dt << Add Multiple Columns("beginning", 2, Before First, Row State);
   // adds two row state columns named "beginning <n>" before the first column

dt = Current Data Table();
dt << Add Multiple Columns("middle", 3, After(:height), Numeric);
   // adds three numeric columns named "middle <n>" after the height column

dt = Current Data Table();
dt << Add Multiple Columns("end", 4, After Last, Character(4));
   // adds four character columns named "end <n>" after the last column
```

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 grouped columns are now height and weight, but the group name still contains age. Once a group is created, its name does not change, even if you remove the first column that was originally grouped.

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 Columns 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"); //selects the columns in the group
dt << Deselect Column Group ("name"); //deselects the columns in the group
```

The following example groups columns X and Y and groups columns Ozone through Lead, and then selects those columns 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( 1 );
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 (col) "name")
```

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" );
```

Select Columns

To select a column, use the `Set Selected` message.

```
col << Set Selected(1);
```

For example, to select all continuous variables in the Big Class.jmp sample data table, use the following script:

```
dt=Open("$SAMPLE_DATA/Big Class.jmp");
cc=dt << Get Column Names("Continuous");
ncols = N Items(cc);
for(i=1,i<=ncols,i++,
   cc[i]<<Set Selected(1);
);
```

Get Selected Columns

To get a list of currently selected columns, use the `Get Selected Columns` message.

```
dt << Get Selected Columns();
   {:age, :sex, :height}
```

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. For more information, see "Column Attributes" on page 323.

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)
```

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 columns to the indicated destination point.

```
dt << Move Selected Columns(To First);
dt << Move Selected Columns(To Last);
dt << Move Selected Columns(After("name"));
```

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"));
```

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.

```
obj << Column Switcher(<default_col>(<col1>, <col2>, ...); //adds a Column
  Switcher to the report
obj << Remove Column Switcher(); //removes the Column Switcher from a report
```

For example, add the Column Switcher to a Contingency report for the Car Poll.jmp data as follows:

```
dt = Open( "$SAMPLE_DATA/Car Poll.jmp" );
obj = Contingency(
  Y( :size ),
  X( :marital status )
);
ColumnSwitcherObject = obj <<
Column Switcher(
  :marital status,
  {:sex, :country, :marital status}
);
ColumnSwitcherObject<<setsize(200); //number of pixels for the switcher width
ColumnSwitcherObject<<setnlines(6); //number of columns to display in
  switcher
```

For more details about the Column Switcher, see the *Using JMP* book.

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 as follows:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Compress Selected Columns(
{:age, :sex, :height, :weight}
);
```

For more information about this feature, see the *Using JMP* book.

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. The following examples use the Big Class.jmp sample data table:

```
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 323 for more information.

Obtain Column Names

`Column Name(`*n*`)` returns the name of the *n*th column. The examples in this section use the Big Class.jmp sample data table.

```
column name(2); // returns 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:

```
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`, or `Row State` to include only those column data types.
- Specify `"Continuous"`, `"Ordinal"`, or `"Nominal"` 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 lock columns(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 uses the Big Class.jmp sample data table, and changes column 2 (age) to ratio. It then returns the current column name to the log.

```
col = Column(2);
col << Set Name("ratio");
col << Get Name;
```

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;    //returns list

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 complete details about value labels, see the *Using JMP* book.

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.

```
:sex << Value Labels({“F”, “M”}, {“Female”, “Male”}); // uses two lists
:sex << Value Labels({"F", "Female", "M", "Male"}); // uses a list of pairs
:sex << Value Labels({"F" = "Female", "M" = "Male"}) // uses 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, proceed as follows:

```
: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.

```
Current Data Table()<<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 to the Big Class.jmp sample data table that has a data type of character.

```
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;
   "Ordinal"
```

You can specify the format of a column when changing its data type as follows:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "Date" );
Column( "Date" ) << Data Type( Numeric, Format( "ddMonYYYY" ) );
```

`Set Data Type` and `Set Modeling Type` work the same as `Data Type` and `Modeling Type`, respectively.

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 fieldwidth(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 130 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 130 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, as follows:

```
col << Get Format;
```

Set, Get, or Evaluate a Formula

To set, get, and evaluate a formula for a column, proceed as follows:

```
col=New Column("Ratio");       // creates column and stores its reference
col << Set Formula(:height/:weight); // sets formula
col << EvalFormula;              // evaluates the formula
col << Get Formula;              // returns 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. Note the following:

- 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 << NewColumn("Ratio",Numeric,Formula(:height/:weight),EvalFormula);
  ```

  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:

  ```
  Current Data Table()<<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 replicable sequence, then you have another reason to use `Run Formulas`, in order to avoid a second evaluation in the background.

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:

```
column("Sex") << List Check({"M", "F"});    // sets it
column("Sex") << List Check();              // clears it
```

Range checks require the specification of a range using the syntax in Table 9.1.

Table 9.1 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 operators 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(), as follows:

```
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;
```

For example, if you sent a Get Range Check for the age column in the example above (where values should be greater than zero and less than 120), the following output appears:

```
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 328 for more information.

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.

```
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 Roles` message. Choices include No Role, X, Y, Weight, and Freq. The `Get Role` message returns the current setting.

```
col = New Column( "New" );
col << Preselect Role( X );
col << Get Role;
```

Lock a Column

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.

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"); // returns 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`.

Table 9.2 shows examples for setting and getting column properties. See The Column Info Window chapter in *Using JMP* for more information on each property.

Table 9.2 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");`

Table 9.2 Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments
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 Ordering	Specifies the order in which you want the data to appear in reports.	`col<<Set Property("Value Ordering", {"Spring", "Summer", "Fall", "Winter"});`
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} )});`

Table 9.2 Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments
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");`
Control Limits	Used for control charts.	`col<<Set Property("Control Limits", {XBar(Avg(44), LCL(29), UCL(69))});` `col<<Get Property("Control Limits");`
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.

Table 9.2 Properties for Data Table Columns *(Continued)*

Property	Description	Example Using 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");
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");

Table 9.2 Properties for Data Table Columns *(Continued)*

Property	Description	Example Using Arguments
`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" ) );`
`Profit Matrix`	Converts a predictive model to a decision model and assigns weights to outcomes. Specify a list that contains the weights in a matrix and categories in another list. In the matrix, each row shows the consequences if you predicted the specified response. Each column shows the consequences if the actual response if the specified response.	`col<<Set Property( "Profit Matrix", {[5 - 1, -1 4, -2 - 2], {"M", "F", "Others"}} );` `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"));`

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

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); // adds 3 rows to bottom of data table
dt<<Add Rows(3, 10); /* adds 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});
add point = expr( dt<<addRows({:xx=x; :yy=y}) );
```

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<<NewColumn("xx",Numeric);
dt<<NewColumn("cc",Character,width(12));

dt<<AddRows({xx=12,cc="Chicago"});                      // single list
dt<<AddRows({xx=13,cc="New York"},{xx=14,cc="Newark"}); // several lists
dt<<AddRows({{xx=15,cc="San Francisco"},{xx=sqrt(256),cc="Oakland"}});
                                                        // list of lists

a = {xx=20,cc="Miami"};
dt<<AddRows(a); // evaluates as single list

b={{xx=17,cc="San Antonio"},{xx=18,cc="Houston"},{xx=19,cc="Dallas"}};
dt<<AddRows(b); // evaluates as list of lists
```

Further details for rows can be specified with messages described under "Row States and Operators" on page 344.

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` deletes the currently selected rows. With neither an argument nor rows selected, `Delete Rows` does nothing.

```
dt<<Delete Rows(10);                        //deletes row 10
dt<<Delete Rows({11,12,13});                //deletes rows 11-13
myList={11,12,13}; dt<<Delete Rows(mylist);//deletes rows 11-13
dt<<Delete Rows(1::20);                     //deletes first 20 rows
dt<<Delete Rows([1 2 3]);                   //deletes 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;dt<<delete rows;  //selects, deletes 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 5 rows from the bottom of a data table:

```
x=5;
n=NRow(dt); For(i=n,i>n-x,i--,
   dt<<Delete Rows(i));
```

`NRow` counts the rows in the table. For more details, see "Iterate a Script on Each Row" on page 342.

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, proceed as follows:

```
dt<<Select Rows({1, 3, 5, 7});
```

To select a range of rows, specify one of the following messages:

```
Current Data Table() << Select Rows( Index( 7, 10 ) );
Current Data Table() << Select Where( Any( Row() == Index( 7, 10 ) ) );
```

Both of these examples select rows seven through 10 in the current data table.

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 90 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 `<< Extend Select Where`. This is an alternative to using an `OR` statement.

```
dt << Select Where( age == 14 );
dt << Extend Select Where( sex == "F" );
```

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;
```

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.

```
New Column("ratio", Formula(height / weight[1]));
```

To obtain a random selection:

```
dt<<Select Randomly(number)
dt<<Select Randomly(probability)
```

These commands use a conditional probability to obtain the exact count requested.

The row menu command **Select Matching Cells** is also implemented in JSL.

```
dt << Select Matching Cells; //selects matching cells in the current data
  table
dt << Select All Matching Cells; //selects matching cells in all open data
  tables.
```

For more complicated selections, or to store selections permanently as row state data, see "Row States and Operators" on page 344.

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`, as follows:

```
dt << Clear Select;
```

Move Rows

These commands move the currently selected rows to the indicated destination point.

```
dt << Move Rows(AtStart);
dt << Move Rows(AtEnd);
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 details about how numbers correspond to colors and markers, see "Colors and Markers" on page 354.

```
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, as follows:

```
dt << Select Where(age==13)   // 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 << Color by Column(:age);
dt << Marker by Column(:age);
```

Additional, named arguments are as follows:

- `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 columns.
- `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:

```
:height << Set Property(
  "Color Gradient",
  {"White to Blue", Range( 40, 80 )}
);
:height << Color Cell By Value( 1 );

:age << Set Property(
  "Value Colors",
  {12 = Red, 13 = Yellow, 14 = Green, 15 = Blue, 16 =
  Magenta, 17 = Gray}
);

:age << Color Cell By Value( 0 ); // turns 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 operators, see "Row States and Operators" on page 344.

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 << Select Where( age > 13 );
dt << Hide(1);
```

Or, because messages to the same object can be stacked together in a single statement, you could simplify it to this:

```
dt << Select Where(age > 13 ) << Hide(1);
```

Iterate on Rows in a Table

In addition to built-in programming operators for iterating, JSL provides operators 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 operators, 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. For more information, see "What is the Current Row?" on page 341.

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. For more information, see "Advanced Scoping and Namespaces" on page 237 in the "Programming Methods" chapter.

You can use the `Row()` operator to get or set the current row number. `Row()` is an example of an *L-value* expression in JSL: an operator 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();   // stores the current row number in x
Row() = 7;   // makes the 7th row current
```

```
Row() = 7; :age;   // makes 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` operators 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 178 in the "Data Structures" chapter for details.

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.

```
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 Rowstate", 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`, as follows:

```
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. For more information, see "Break and Continue" on page 108 in the "JSL Building Blocks" chapter.

Return Row Values

`Dif` and `Lag` are special operators 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.

Example:

```
// Create a new data table
dt = New Table("Sequence Example");

// Add 2 columns and 50 rows
dt << New Column("Count to Five");
dt << New Column("Count to Seventeen by Fours"); dt << Add Rows (50);

/* 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 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`.

Row States and Operators

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 operators to manipulate row states.

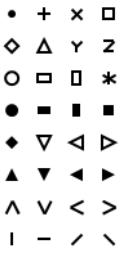
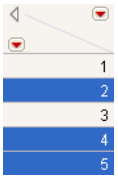
About Row States

Row states change how JMP works with your data. Table 9.3 explains each row state. Remember that you can use several row states at once to get the combination of effects that you want.

Table 9.3 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`.)

Table 9.3 Row States *(Continued)*

Row states	How they affect results
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.
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.

About Row State Operators

When you give a row state operator a number as its argument (or an expression that evaluates to a number), the operator interprets that number as an index to its possible values.

Table 9.4 shows a comparison chart of the different row state operators, so that you can see which operators convert row states to numbers and numbers to row states. It also includes the numbers that you can use with each operator.

Table 9.4 Row State Operators

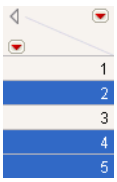
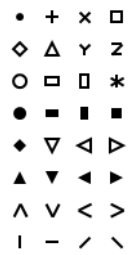
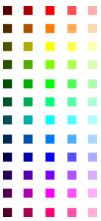
Numbers	Convert from numbers to row states →	Row states	Convert from row states to numbers →	Numbers
1 or 0	`Excluded State(`*n*`)`	Excluded 	`Excluded(`*rowstate*`)`	1 or 0
1 or 0	`Hidden State(`*n*`)`	Hidden 	`Hidden(`*rowstate*`)`	1 or 0
1 or 0	`Labeled State(`*n*`)`	Labeled 	`Labeled(`*rowstate*`)`	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 9.4 Row State Operators *(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 the *Using JMP* book.

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" );
```

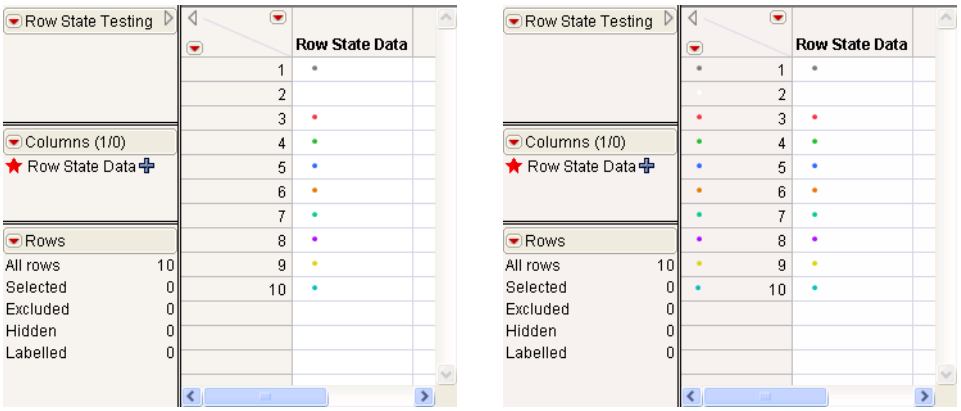
2. Submit these lines to add row states to a column and add 10 rows:

```
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 9.6 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 details about how to change selected attributes of a row state without changing or canceling others, see "Set One Characteristic and Cancel Others" on page 352.

Set or Get Row States

From JSL, you can set or get row states directly using the `Row State` operator. 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 ); // makes row 1 red
Row() = 8; Row State() = Color State( 3 ); // makes the 8th row red
For Each Row( Row State() = Color State( 3 ) ); // makes each row red
```

To specify the current 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 ); // makes row 1 red;
```

To get a row state, place the row state expression on the right side of an assignment:

```
x = Row State( 5 );  // puts 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:

```
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; // makes row 1 red without changing marker
```

And setting every aspect of a row state to a single state:

```
Row State( 1 ) = Color State( 5 ); // makes row 1 blue and removes its marker
```

To copy all the current row states into a row state column:

```
New Column( "rscol", Set Formula(Row State() );
For Each Row( rscol = 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 operators convert numbers into states, and others convert states into numbers. Here are some helpful hints for remembering which are which:

Number-to-state operators have the word "State" Operators 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 operators are one word or have the word "Of" Operators that take row state arguments (and assume that the argument `Row State()` if none is given) and operators 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 9.4 on page 346 is a helpful comparison chart for these operators.

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 = New Table( "Row State Testing" );
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
  Color State(1)

Row() = 8; Row State();  // returns row state for current (8th) row
  Color State(8)

For Each Row( Print( Row State()) ); //returns the row states for each row
  Color State(1)
  Color State(2)
  Color State(3)
  Color State(4)
  Color State(5)
  Color State(6)
  Color State(7)
  Color State(8)
  Color State(9)
  Color State(10)
```

To get a row state and store it in a global, place `Row State()` on the right side of an assignment, as follows:

```
::x = Row State(1); // puts the row state of row 1 in x, a global

Row() = 8; ::x = Row State();  // puts the row state of the 8th row in x
Show(x);
  x = Color State(8)

dt << New Column( "rscol", Row State) ;
For Each Row( :rscol = Row State() );
// put row states in rscol (a row state column)
```

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 uses Big Class.jmp. Set green Y markers for males, but hide them in plots for now. Set red X markers for females and do not hide them in plots.

```
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` operator for getting or setting all the characteristics of a row state, there are separate operators 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 operators that set one characteristic and cancel others are as follows: `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 = Current Data Table() << 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; // excludes current row
Hidden()=0;              // unhides current row
Labeled(Row State())=1;  // labels current row
Selected()=0;            // deselects 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); // changes current row state to excluded
Row State()=Hidden State(0);   // changes current row state to not hidden
Row State()=Labeled State(1);  // changes current row state to labeled
Row State()=Selected State(0); // changes 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`:

```
Row State()=Combine States(Color State(4), Marker State(3), Hidden State(1));
// changes the current row to hidden and a green square marker
```

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 << New Column("myExcl",Row State, set formula(Excluded
   State(Modulo(Row(),2))));
For Each Row(Row State()=Combine States(:myExcl, Row State()));
```

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;           // sets 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; // sets 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);  // changes current row to green
Row State()=Marker State(4); // changes 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:

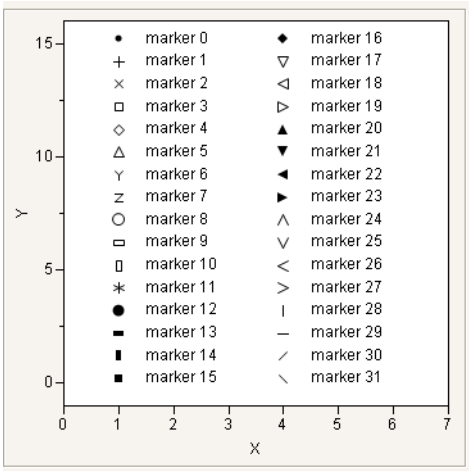
```
Row State()=Combine States(Color State(4), Marker State(3), Hidden State(1));
// changes current row to hidden with green square marker
```

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 9.7 JMP Markers



Tip: For more details about this script, see the "Display Trees" chapter on page 401.

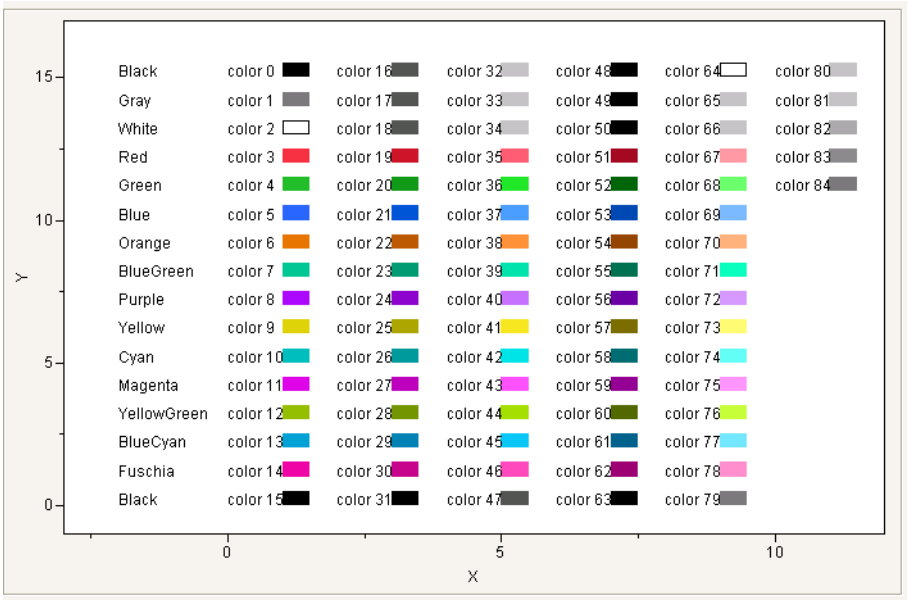
Note the following about colors:

- 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 "Colors" on page 516 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 9.8 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

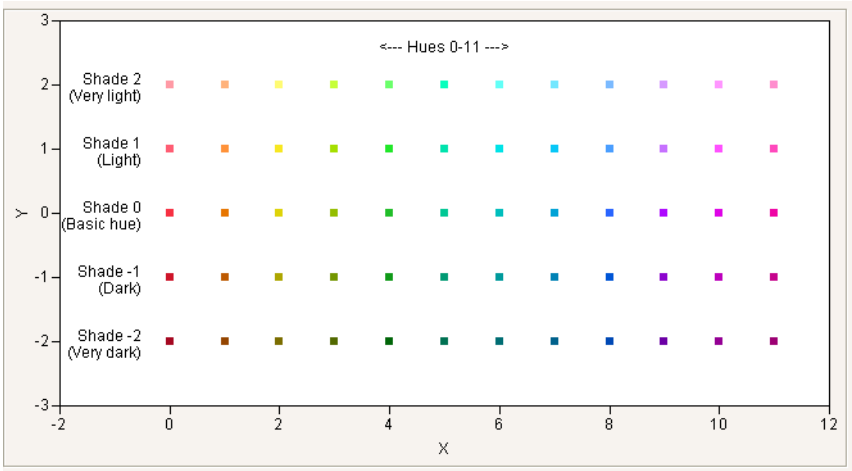
`Hue State` and `Shade State` together are an alternative to `Color State` for choosing colors. You cannot select black, white, or the shades of gray when you use `Hue State`. For these, you must use `Shade State` alone, or `Color State`.

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 9.9 Hues and Shades



There are no `-Of` operators 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.

```
// assumes CP and CA data such as simulated below
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} );
       )
    )
);
```

The Numbers behind Row States

This section is an *optional topic for advanced users* who are interested in working with row states through their internal numeric codes.

A row state is a collection of six attributes that all rows in a data table have. These six attributes are packed into a single number internally. You can see row states' internal coding if you want. Simply copy row states to a column, and then change the column's type to numeric to see the numbers that JMP uses.

It is also possible to assign row states through their internal numeric codes using the `As Row State` operator, which simply converts integers to their equivalent row states. For example, to assign row states according to the row number, you could do:

```
For Each Row(Row State()=as row state(row()));
```

In addition, the `Set Row States` command enables you to submit a matrix of codes that assign the row states all at once. The matrix should have dimension (number of rows) by 1, and

contain one entry for each row. The entries are the row state codes corresponding to the row's desired state.

Such row states are unlikely to be of any use, however. For practical applications, understanding how numbers are related to row states is important. Briefly, for some row state *r*, such as the row state of the 3rd row as shown here, the row state code is computed by this formula:

```
r=Row State(3);
rscode=selected(r)+
  2*excluded(r)+
  4*hidden(r)+
  8*labeled(r)+
  16*marker of(r)+
  256*color of(r);
```

Example

This example takes advantage of this method to develop a compact formula that distinguishes females and males with *X*s and *Y*s, while excluding females from calculations, hiding males from plots, and assigning different colors for each age.

Recall that the logical == operator is an equality test that returns 1 for true and 0 for false.

```
dt=open("$SAMPLE_DATA/Big Class.JMP");

For Each Row(
  Row State() = as row state(
    (sex=="F")*2 + // exclude F
    (sex=="M")*4 + // hide M
    ((sex=="F")*2+(sex=="M")*6) * 16 +
      // marker 2 for F, 6 for M
    (age-11)*256));
      // colors 1 through 6
```

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 290.
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 361.

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. For the following example, use `Big Class.jmp` and select row 5 to be the current row.

```
Row() = 5;
dt:age = 19; // sets the age in row 5 to 19
dt:name = "Sam"; // sets the name in row 5 to Sam
```

To set the value of a cell in another row, 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 the current row, specify the column name. Suppose that row 16 in Big Class.jmp is selected in the following examples.

```
Row() = 16;
myGlobal = :age;
   14
:age;
   14
Show(:age);
   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. For more information, see "Access Cell Values through Column References" on page 314.

To summarize:

- 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 the `:` infix operator and a subscript, as follows:

```
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
```

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.

```
Current Data Table() << Begin Data Update;
...<many changes>...
Current Data Table() << 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.

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. Metadata includes the following:

- Table variables (store text strings, such as notes)
- Properties (store expressions, such as scripts)
- Scripts
- Formulas

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` or `New 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");
```

or

```
dt << New 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("$SAMPLE_DATA/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 = Current Data Table();
dt << New Script("Bivariate", Bivariate(Y(weight),X(height)));
dt << Get Script("Bivariate");
   Bivariate(Y(weight),X(height))
dt << Set Property("Bivariate",Bivariate(Y(weight),X(height),Fit Line));
dt << Run Script("Bivariate")
```

The following example creates a new script and then runs the script:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Script(
   "New Script",
   Distribution(
      Column( :Height, :Weight ),
      By( :sex )
   )
```

```
);
dt << Run Script( "Distribution" );
```

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, 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 ) )
         )
      ),
```

A table variable is counted as a script in a data table. `Run Script(2)` runs the first script in a data table that contains a table variable. For more predictable results, refer to the script by name rather than number.

Automatically Run a Script upon Opening a Data Table

The table script named `On Open` or `OnOpen` can be automatically run when the data table opens. Users are prompted to run the script by default. Their choice is remembered each time they 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, and then double-click the property name and change the name to `On Open`.
- Store the script using a `New Script` message.

In this example, you send the `Sort` message to `Current Data Table()` rather than *dt*, because *dt* might not be defined when the data table is opened.

```
dt << New Script("OnOpen", sortedDt=Current Data Table() << Sort( By(Name),
  Output Table Name ("Sorted Big Class")) );
```

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.

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 `EvalFormula` command to the column. You can even do this inside the command to create the column, after the formula clause, as follows:

```
Current Data Table() << New Column("Ratio", Numeric, Formula(:height/
  :weight), EvalFormula)
```

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), and formulas from the data table, using the following commands:

```
dt << Delete Table Variable(name);
dt << Delete Table Property(name); //
col << Delete Formula;
col << Delete Property(name);
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 296). 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:

```
// scalar result for row 7 assigned to JSL global variable
row()=7; ::scalar = Mean(height, weight);
```

```
// formula column created in data table
New Column("Scaled Ht-Wt Ratio",
   formula(mean(height, weight)/age));

// vector of results
vector=J(1,40); // create a 1x40 matrix to hold results
For Each Row(vector[row()]=mean(height,weight)); //fill vector
```

Rowwise functions can also take vector (column matrix) or list arguments, as follows:

```
myMu=mean([1 2 3 4]); mySigma=stddev({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(...)`.

Chapter 10

Scripting Platforms

Create, Repeat, and Modify Analyses

Platforms can be launched from scripts and subsequently controlled from scripts. If you do an analysis interactively, you can save a script that recreates the analysis.

This chapter describes how to write scripts for platforms and provides tips on platform-specific commands. For additional information, see the JMP Scripting Index. (Select **Help > Scripting Index** in JMP.)

Contents

Overview

Learning how to script platforms is no harder than learning how to control platforms interactively. Writing scripts yourself takes some effort, since you have to type them and get the syntax right. Fortunately, you can have the platform write much of a script for you, and the additional work is well worth it if you need to automate repetitive production analyses. Also, scripting presents valuable opportunities to customize and combine analyses.

Be aware that the script-saving feature of platforms has some limitations. JMP saves analytical commands and most (but not all) visual customizations. Therefore, if your visual customizations are important to you, you might occasionally need to learn to program the display interface. JMP records the state of an analysis, but not the sequence of events leading to that state. If you want scripts to play back demonstrations of JMP platforms in action, you have some learning to do.

Platform results exist in two layers: the platform itself, containing the analytical results and responding to analytical commands; and the presentation display, which responds to a different set of commands. A third object is the data table itself, which can be involved in a live analysis.

1. If you want to add to the analysis, such as to fit a new line, you send a command to the platform itself using techniques discussed in this chapter.
2. If you want to make the frame of a graph larger, you send a command to the display. See "Manipulating Displays" on page 403 in the "Display Trees" chapter.
3. If you want to highlight certain points representing rows in a data table, you send a row state command to the data table. See "Row States and Operators" on page 344 in the "Data Tables" chapter.

If you are interested in building your own custom analysis platforms and reports, see "Constructing Display Trees" on page 423 in the "Display Trees" chapter. If you would like to build your own custom statistical calculations using JSL's compact matrix notation, see the "Matrices" on page 173 in the "Data Structures" chapter.

Scripting Analysis Platforms

Scripting platforms is done with the same keywords that you see in the windows and menus, including the choices that you see in red triangle menus and context menus inside a report.

You can use JMP interactively and then obtain the script corresponding to the analysis that you created. You can take advantage of this behavior in several ways:

1. Learn JSL by example. Launch platforms and work with them interactively, and then save and examine the script that matches your work. Make changes through the interface and then study the changes in the script.
2. Save a number of scripts from analyses to a script window, and then save it to a file. You can use that script to recreate the analysis later, or to reproduce the analysis using new data.
3. Save platform scripts, and then edit them to use JSL programming features to make the analyses more general and to customize reports.

Select **Help > Scripting Index**, and then select **Objects** from the menu to see all the scripting commands available for each platform. This index includes all scriptable platform options, with a description, the syntax, and an example.

Notation in this chapter

This chapter `Capitalizes The Names` of command words that you need to use exactly as they are shown and shows arguments that are placeholders for actual choices in `lowercase`. For example, `Connect Color` is a command 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} );
```

You do not have to use upper- and lower-case letters, and you do not have to type the `|` character either.

You can use the following terms interchangeably depending on context: command, option, or message. The term argument refers to anything that goes in parentheses after an item, and arguments can also contain options with further arguments.

Launching Platforms Interactively and Obtaining the Equivalent Script

Here is an example to see how the JMP interface and scripting language are related.

Launch a Platform

1. Select **Help > Sample Data Library** and open Big Class.jmp.
2. Select **Fit Y by X** from the **Analyze** menu.
3. Select height and click **Y, Response**.
4. Select age and click **X, Factor**.
5. Click **OK**.

Save Script

All analysis platforms have a **Script** submenu with the commands shown here to create a JSL script that duplicates the analysis in its current state. These commands are choices for where to send that script.

Redo Analysis Launches a new copy of the platform, rebuilding all the analyses as specified. This is useful for updating your results when the state of the data table has changed. For example, after you fix errors in the data table, select a subset, add more data, and so on.

Relaunch Analysis Opens the launch window, already filled in with what is needed to produce the report.

Automatic Recalc Turns Automatic Recalc on or off.

Copy Script Copies the script that reproduces the report onto the clipboard. You can then paste the script into a script window or another application.

Save Script to Data Table Saves the script as a new property in the data table. In the data table window, you see the script named for the platform, and it has a pop-up menu to let you **Run Script**. Properties such as saved scripts are saved with the data table for later use. Many sample data tables have example scripts.

Save Script to Journal Saves a button to the current journal (or a new one, if a journal is not open) that runs a script to reproduce the report.

Save Script to Script Window Presents the script for the current state of the platform in a script window, where you can view, edit, and submit it.

Save Script to Report Saves the script in a text box at the top of the report. The script to reconstruct an analysis is available in any journals that you might save, as well as in the

platform window itself. This also serves as a concise overview, listing all the report's analyses and their argument settings.

Save Script for All Objects Saves the script for all objects within a multi-platform report in a script window, where you can view, edit, and submit it. An example is Fit Y by X, where various combinations of continuous, nominal, and ordinal columns result in Bivariate, Oneway, Contingency, and Logistic platform objects all within a single report. `Save Script to Script Window` saves only the script for the single object from which you selected the option. `Save Script for All Objects` saves the script for all objects within the window, regardless of which object's menu option you used. Another example is that if you have By groups, `Save Script for All Objects` saves the script for all the BY group objects within the window, whereas `Save Script to...` captures the script for only the one group. All the BY groups appear in the same window when the report is re-run, because `Save Script For All Objects` wraps all By-variable analyses in a `New Window()` command. For example, a bivariate fit of height by weight grouped by males and females would result in the following script.

```
New Window("Big Class.jmp: Bivariate",
Bivariate(Y( :height), X( :weight), Fit Line, Where( :sex == "F"));
Bivariate(Y( :height), X( :weight), Fit Line, Where( :sex == "M")));
```

Save Script to Project Creates a script to reproduce the report and saves it as a JSL file in a JMP project.

Data Table Window Displays a data table window with the BY group data associated with the analysis when it has By groups, and brings the data table window forward otherwise.

Make Some Changes

Now, continue working in the Oneway report window.

1. From the red triangle menu for Oneway Analysis of height by age, select **Means/Anova**.
2. From the same menu, select **Compare Means > Each Pair, Student's t**.
3. From the same menu, select **Script > Save Script to Script Window**.

The resulting script is as follows:

```
Oneway(
   Y( :height ),
   X( :age ),
   Each Pair( 1 ),
   Name( "Means/Anova" )(1),
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
```

The script records the choices that you made in the menu. Since these are all Boolean (on or off) options, they have an argument of 1 to turn them on. In addition to the options selected, three additional display options have been implied from the other commands given. Choices that you make in menus and their corresponding JSL commands have exactly the same effects.

You could submit this script to get exactly this report quickly, without all the interactive steps. To run a script from a script window, select the text and then either select **Edit > Run Script**, or type CONTROL-R (Windows) or COMMAND-R (Macintosh). If you select no text, the entire window is run.

Syntax for Platform Scripting

Look at the script in more detail.

```
Oneway(
   Y( :height ),
   X( :age ),
   Each Pair( 1 ),
   Name( "Means/Anova" )(1),
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
```

All platform scripts start with a command to call the platform, in this case `Oneway`. Inside the `Oneway` command are two types of arguments: arguments like the Y and X column role lists are required at launch, and options like `Each Pair(1)` are sent to the platform after it is launched.

Most options are simple on/off choices with check marks (or not). The scripting equivalent of a check box is a Boolean argument 1 or 0.

Other commands lead to dialog boxes where you specify values or make choices. In scripts, such specifications are given inside parentheses and separated by commas, usually in the same order as they appear in the window (top to bottom, left to right).

BY Group Reports

In most platforms, you can run the platform repeatedly across subgroups of the rows as defined by one or more by columns. To do this in scripts, include a `By` argument in the launch command, listing each column as the argument to `By`.

Example

1. Select **Help > Sample Data Library** and open Big Class.jmp.
2. Run this script to produce a bivariate report by sex:

```
biv = Bivariate(Y(weight), X(height), By(sex));
```

This launch message produces a report window with two nodes in the outline, one for the rows of the data table where sex is "F" and a second for rows where sex is "M." Rather than returning a reference to a platform, `Bivariate[]`, the platform object returns a list of references `{Bivariate[], Bivariate[]}`:

```
show(biv);
   biv = {Bivariate[], Bivariate[]}
```

You can direct messages either to each reference individually or to the list of references, depending on whether you want to change selected nodes individually or all nodes simultaneously.

3. Run this line to add a regression fit to both nodes:

   ```
   biv << fit line;
   ```

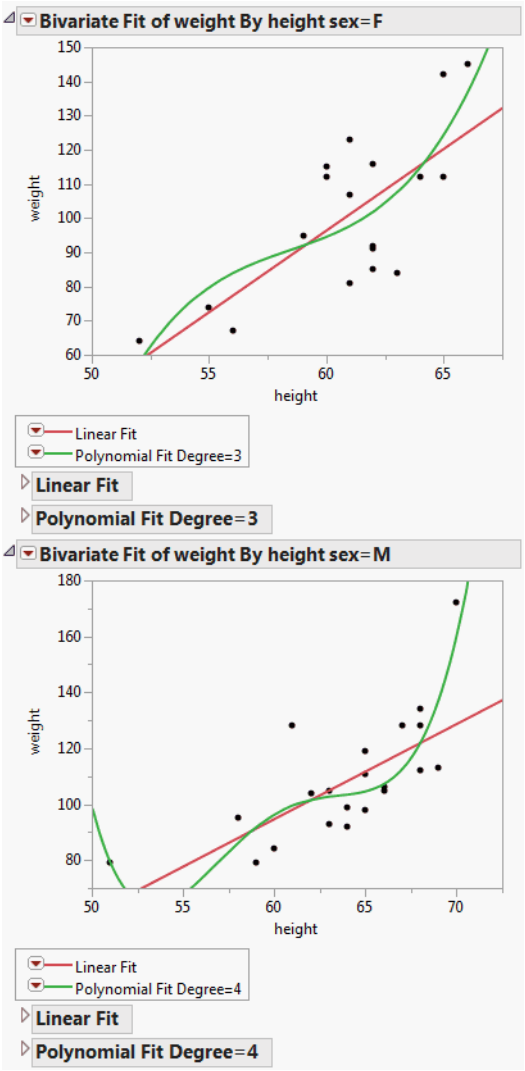
4. Run this line to add a cubic fit to the F node:

   ```
   Bivariate[1] << fit polynomial(3);
   ```

5. Run this line to add a quartic fit to the M node:

   ```
   Bivariate[2] << fit polynomial(4);
   ```

Figure 10.1 Accessing By Group Reports



Multiple columns listed for `By()` produce nodes for each subgroup. For example, `By(sex, age)` would produce nodes for females age 12, females age 13, ..., females age 17, males age 12, males age 13, ..., and males age 17.

The following shows how to launch a platform with `BY` groups and extract results from each group:

```
// open data table Big Class
dt=Open("$SAMPLE_DATA/Big Class.jmp");

// launch Oneway platform
```

```
onew = Oneway(x(age),y(height),by(sex),anova);
                  // onew is a list of platform object references
r = onew<<report; // r is a list of display boxes
nBy = nItems(r);  // the number of by groups
vc = j(nBy,1,0);  // a place to store the variances

// now extract results
for(i=1, i<=nBy, i++,
   vc[i] = r[i][
      OutlineBox("Analysis of Variance"),
      ColumnBox("Sum of Squares")][2]);
show(vc);

summarize(byValues=by(sex));
newTable("Variances")
   << newColumn("Sex",character,width(8),values(byValues))
   << newColumn("Variance",numeric,"continuous",values(vc));
```

Saving BY Group Scripts

In addition to the Script submenu in an analysis platform, a **Script All By-Groups** submenu appears when appropriate. This submenu lets you save scripts to reproduce a report created with By groups. It contains the following options:

- **Redo Analysis**
- **Relaunch Analysis**
- **Copy Script**
- **Save Script to Data Table**
- **Save Script to Journal**
- **Save Script to Script Window**

All of these work just like their counterparts on the Script submenu, except they reproduce all By groups in a report.

Note: To save a script to a data table, journal, or script window, use the `Save ByGroup` function. See the JMP Scripting Index for examples.

Sending Script Commands to a Live Analysis

After the platform has been launched, there is a different syntax to send messages to control the live platform, using the `Send` function or its operator equivalent, `<<`.

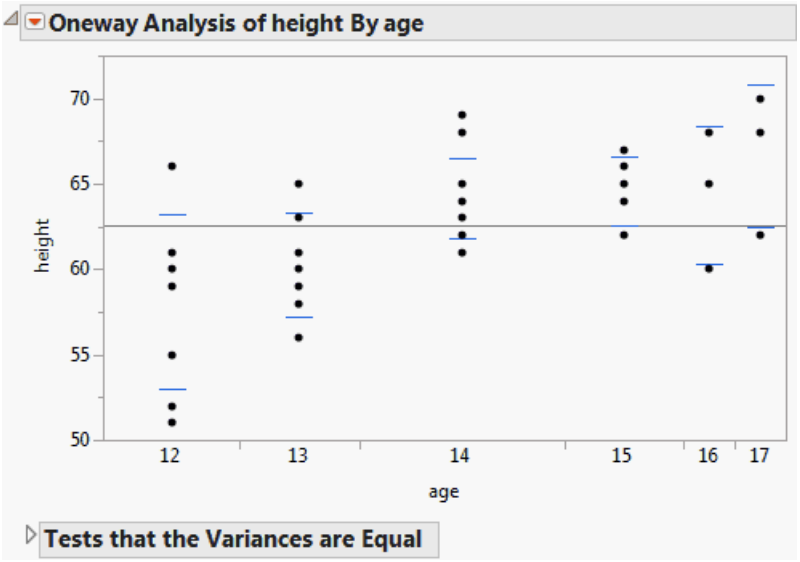
First you need a way to address the object. The simplest way is to launch the platform from a script that assigns a reference to the object to a global variable. For example, this script saves a reference to a Oneway analysis in the variable *oneObj*:

```
oneObj = Oneway(Y(height), X(age));
```

Then, you use either the Send function or the equivalent << operator to send a message:

```
Send(oneObj, Unequal Variances(1));
oneObj << Unequal Variances(1);
```

Figure 10.2 Adding Elements to a Report Using a Script



Conventions for Commands and Arguments

1. For most messages, omitting the argument for a Boolean command enables the option. The following commands 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" );
   ```

   For row state messages, omitting the argument *toggles* the option: if the option is off, the message turns it on. If the option is on, the message turns it off. You can also use the "toggle", "switch", or "flip" keyword, as in `r << Exclude( "toggle" )`.

2. In cases where the menu gives several options separated by comma or slash, as in the case of **Means/Anova/t Test** above, you can use any one of the commands. In cases where

several commands have the same alias, as in the two cases of "Means" above, the first one takes precedence in the scripting language.

3. If you make changes to the display, such as resizing the graph, they are also saved in the saved script.
4. If there are submenus whose items represent commands rather than settings, then the corresponding script is the items themselves without the parent item. For example, above see that Oneway has a menu item **Nonparametric** with three commands in a submenu, including `Wilcoxon Test`. You would use just the subitem name in scripts, for example:

   ```
   oneObj = Oneway(Y(height), X(age), Wilcoxon Test(1));
   oneObj << Wilcoxon Test(1);
   ```

5. If there are submenus whose items are values for a setting rather than independent commands, then in script, you give the parent item with the submenu choice as its argument. For example, Oneway has a submenu for `Set Alpha Level`, whose choices are `0.10`, `0.05`, `0.01`, and `Other...` To change the value in script, you give your choice as an argument to `Set Alpha Level`:

   ```
   oneObj << SetAlphaLevel(0.01);
   ```

6. The script returned from a platform often looks different from a script that you write yourself. For example, you could launch Distribution with this brief script:

   ```
   dist=Distribution(Y(Height,weight));
   ```

But if you then ask Distribution to save its script, you see considerably more detail:

```
Distribution(Continuous Distribution(Column( :height), Axis
   Settings(Scale(Linear), Format("Fixed Dec", 0), Min(50), Max(72.5),
   Inc(5), Minor Ticks(1))), Continuous Distribution(Column( :weight), Axis
   Settings(Scale(Linear), Format("Fixed Dec", 0), Min(70), Max(180),
   Inc(10))));
```

Sending Several Messages

To send several messages, you can 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 up multiple messages with extra << symbols to perform them all in order (left to right). You can use grouping parentheses to send a message to the result of sending a message:

```
(dist<<stem and leaf(1)) << horizontal layout(0);
```

In this case, the associative grouping is of no consequence, because messages are performed left-to-right anyway. However, a case where it *would* matter is when sending messages to child objects, discussed next.

Another way to stack messages is to send a list of messages:

```
dist<<{quantiles(1), moments(1), more moments(1), horizontal layout(1)};
```

Learning the Messages an Object Responds to

Now that you have an object, what messages can you send it? There are several ways to learn your options:

1. Try the procedure through the interactive interface first, then study the saved script.
2. Study the interface in the platform window. Items in the pop-up menus and context menus all have JSL equivalents with the same names and arguments, as discussed under "Launching Platforms Interactively and Obtaining the Equivalent Script" on page 373.
3. Go to the **Help** menu, select **Scripting Index**, find the object type that you are interested in, and click the item in the list.
4. `Show Properties(objectRef)` lists to the Log window all messages the object can receive:

```
show properties(oneObj);
   Quantiles [Boolean](Shows or hides a quantile report.)
   Means/Anova [Boolean](Shows or hides both an ANOVA and a means report.)
   Means/Anova/Pooled t [Boolean](Shows or hides a t test, an ANOVA, and a means
      report.)
   Means and Std Dev [Boolean](Shows or hides a report with both the mean and
      standard deviation for each level of the X variable.)
   ...
```

`Show Properties` 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 278 in the "Data Tables" chapter and "Learning What You Can Do with a Display Box" on page 412 in the "Display Trees" chapter.

How to Interpret the Listing from Show Properties

Notice that most items in the `Show Properties` output have hints inside brackets [ ] at the end of each line. This section examines the `Show Properties` for Bivariate as a general example.

```
biv=Bivariate(Y(height), X(age));
show properties(biv);
```

`[Subtable]`s refer to a set of commands that are put in a submenu. The commands in the subtable are indented, and you use the subitem 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.)
   ...
```

[Boolean]s turn an option on or off, and their arguments are usually 1 or 0. If specified without an argument, sending the message flips it to the opposite state. Frequently [Boolean] messages also indicate that they by [Default On].

```
Show Points [Boolean] [Default On]
```

[New Entity] is a command that leads to a new window in the user interface. Here's an example from the Distribution object properties:

```
Prediction Interval [New Entity] (Prediction Interval to contain a single
   future observation or the mean of m future observations.)
```

[Action]s are choices the user makes in the user interface. They do not have a specific standard for their arguments, so try the item in the interface first and then study the script that the platform saves.

```
Fit Mean [Action] (Fits a flat line at the mean.)
Fit Line [Action] (Fits a regression line to the data.)
```

[Action Choice]s and [Enum]s have a specific set of choices for their arguments.

```
Fit Polynomial  [ActionChoice]  {,2,quadratic,,3,cubic,,4,quartic,5,6}
```

Here's an example of [Enum] in the Bubble Plot properties:

```
Draw [Enum] {Filled, Outlined, Filled and Outlined}
```

Confusion alert! Do not confuse a reference to a *platform* with a reference to a *report*. They are different types of objects and can receive different types of JSL messages. For example, platforms can do such things as run tests, draw plots, or close entire windows. Reports can do such things as copy pictures, select display boxes, or close outline nodes.

This chapter discusses how to script platforms. To learn how to script reports, see the section "Manipulating Displays" on page 403 in the "Display Trees" chapter.

Launching Platforms

Specifying Columns

Scripts to launch a platform should generally specify the columns to analyze. If you submit a script that launches a platform without specifying columns and roles, you get the dialog box for launching the platform. After you choose columns and click **OK**, you get the analysis that you specified by script. In other words, JMP remembers and obeys any other messages in your script after getting the column assignments that it needs.

If you want to use an expression to be evaluated for column arguments, put the names inside an Eval or EvalList function.

```
Distribution(Y(Eval("X"||char(i))));
```

Column arguments for platform launch scripts can also be lists with braces { }, so the following are all valid:

```
Distribution(Y(height,weight));
Distribution(Y({height,weight})); // equivalent
Distribution(Weight({})); // empty specification
                          // (presents the platform's launch window)
```

Throughout this manual, a *col* placeholder represents any data table column reference, and a *nomCol*, *ordCol*, or *contCol* placeholder suggest that a nominal, ordinal, or continuous column, respectively, would be most appropriate. In many cases, columns of other modeling types would also be accepted. JMP returns an error if you try to cast a column into a role that is strictly not allowed.

Column References and Where Statements in Platform Scripts

Creating a column reference enables you to access the values in that column. Store each column reference in a global variable. You can then use the variables instead of column names in the Y and X roles and elsewhere in the script.

Platform scripts that include a `Where` statement treat column references differently. You must use `Eval()` on the column reference variables to return the column name. The other option is to use only column names.

This example defines column references for the weight and height columns. The Bivariate platform message is passed to the data table, and the column references are specified as Y and X roles. Notice the `Eval()` statements that get the column names for *Ycol* and *Xcol*.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Ycol = Column( "weight" );
Xcol = Column( "height" );
biv2 = dt << Bivariate(
   Y( Eval( Ycol ) ), // get column names for Ycol and Xcol
   X( Eval( Xcol ) ),
   Fit Line(),
   Where( :sex == "F" ) // select rows in which sex is F
);
```

When the platform script selects a subset of the data, the column names are now correctly resolved to the original data table. The platform recognizes that the subsetted data and the original data table are related.

Platform Action Command

The command `Action`, if sent to a platform, simply evaluates the expression, whatever it is. You can use this command to chain invocations to platforms where you want to use the platform launch window to ask the user to choose columns and then continue the script after the platform is launched.

In the following example, the script first asks you to assign columns for each of the four platforms in turn, and then print `Done` to the log. The script is effectively stopped four times, each time to prompt for columns for a platform launch. At each step, the user fills in a launch window. Four reports are open at the completion of the script.

This script runs to completion when the first platform launch window is brought up. The other behaviors are run from the stored expressions.

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
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(
  Print("Done");
);
```

Invisible Reports

Platform launches have an `invisible` option, which suppresses the showing of the window. Using this option on a Fit Model script suppresses both the model launch window and the results window.

When using this option, be careful to keep track of the window in the script and close it when the script is done with it. The invisible windows use resources that must be manually freed.

The following example extracts the *F*-Ratio from a bivariate report. Make sure Big Class.jmp is open when running this script.

```
biv = bivariate(x(height),y(weight),invisible);
biv<<fit line;
r = biv<<report;
fratio = r[ColumnBox("F Ratio")][1];
r<<close window;
```

The `invisible` option also works on the **Tables** menu operations, suppressing the creation of the window (rather than just hiding it).

```
Current Data Table()<<Select Where(:age==14);
subDt = Current Data Table() << Subset(invisible);
subDt<<bivariate(x(height),y(weight),fit line);
```

Note that the above script could more easily be done by a `WHERE` clause in the `Bivariate` command.

```
bivariate(x(height),y(weight), Where(:age==14),fit line);
```

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. For example, the following replaces the standard bivariate report's title with a user-specified one.

```
Bivariate(x(height),y(weight), title("my title");
```

General Messages for Platform Windows

The `Save Script` commands are applicable to almost all of the platforms. Some additional commands are shown in Table 10.2 "Messages that can be sent to platform windows" on page 386. In particular the `Report` command lets you access the display surface to control appearance details such as window zooming, scrolling, and so on. For example, to close an outline node for a report, you first obtain a reference to the Display Box tree, subscript to navigate to the outlineBox, and then send it the close message:

```
r = platformRef << Report;
r["Summary of Fit"] << close;
```

This is discussed in greater detail in "Manipulating Displays" on page 403 in the "Display Trees" chapter.

Table 10.1 Scripting Analysis Platforms

Goal	Command	Explanation
See available messages	`Show Properties(`*`obj`*`)`	Shows the messages that a given object can interpret, along with some basic syntax information. Works with all scriptable objects, not just platforms.
Send message	*`obj`* `<<` *`message`* `Send(`*`obj`*`, `*`message`*`)`	Sends a *`message`* to a platform *`object`*.

Table 10.1 Scripting Analysis Platforms *(Continued)*

Goal	Command	Explanation
Send several messages	*obj* << *message* << *message*... *obj* << {*message*, *message*, ...}	Sends a series of *messages* (in order, left to right) to the platform *object*.
Send message to child	*obj* << (*child* << *message*)	Sends a *message* to a *child* object within a platform *object*.
Suppressing output	Platform name(...,invisible)	Keeps output from a report from displaying on-screen. With Fit Model, both the model window and report are suppressed.
Changing a report's title	Platform name (...,title("*string*"))	Changes the report title to *string*. The By variable is appended to the title when necessary.
Changing the automatic recalc setting	Platform name (...,automatic recalc(*Boolean*))	True (1) sets automatic recalc on, so that any changes in the data or in the excluded state are immediately reflected in the report. False (0) sets automatic recalc off, so you must use redo analysis to see the changes. **Note:** If automatic recalc is on, you should use wait(0) commands to let the triggers to take effect and do the recalculation.
Suppressing the current platform preference settings	Platform name (Ignore Platform Preference (*Boolean*))	True (1) ensures that the current platform preference settings are not applied. False (0) means that the current platform preference settings will be applied.

Table 10.2 Messages that can be sent to platform windows

Message	Syntax	Explanation
Redo Analysis	*obj*<<Redo Analysis	Launches the platform again with the same options.
Save Script to Data Table	*obj*<<Save Script to Data Table	Saves script to reproduce analysis as a property in the associated data table.

Table 10.2 Messages that can be sent to platform windows *(Continued)*

Message	Syntax	Explanation
Save Script to Report	*obj*<<Save Script to Report	Saves script to reproduce analysis as a text box at the top of the report.
Save Script to Script Window	*obj*<<Save Script to Script Window	Saves script to reproduce analysis in the Script Journal window.
Save Script for All Objects	*obj*<<Save Script for All Objects	Saves script to reproduce all analyses found within the object's window in the Script Journal window.
Get Script	*x* = *obj*<<Get Script	Returns script to reproduce the analysis as an expression.
Data Table Window	*obj*<<Data Table Window	Makes the associated data table window active (frontmost).
Close Window	*obj*<<Close Window	Closes the window identified by the *obj,* typically a platform surface.
Move Window	*obj*<<Move Window(*x*, *y*)	Moves the window to the (x,y) location on your screen.
Show Window	*obj*<<Show Window(1)	Shows the identified window.
Zoom Window	*obj*<<Zoom Window	Resizes the window to the maximum size of its contents.
Scroll Window	*obj*<<Scroll Window(*x*, *y*) *obj*<<Scroll Window({*x*, *y*})	Scrolls the window *x* pixels to the left and *y* pixels down from the current position; negative coordinates go right and up. If the coordinates are a list in braces { }, they are absolute coordinates. The window scrolls to the point *x* pixels from the left and *y* pixels from the top.
Bring Window to Front	*obj*<<Bring Window To Front	Brings the identified window to the front.
Size Window	*obj*<<Size Window(*x*, *y*)	Resizes the window to *x* pixels wide by *y* pixels high.

Table 10.2 Messages that can be sent to platform windows *(Continued)*

Message	Syntax	Explanation
Maximize Window	*obj*<<Maximize Window	Maximizes the window; equivalent to clicking the maximize button in the upper right corner of the window. This message takes an optional Boolean argument: `// maximize the window:` `obj<<Maximize Window(1)` `// restore the window:` `obj<<Maximize Window(0)`
Minimize Window	*obj*<<Minimize Window	Minimizes the window; equivalent to clicking the minimize button in the upper right corner of the window. This message takes an optional Boolean argument: `// minimize the window:` `obj<<Minimize Window(1)` `// restore the window:` `obj<<Minimize Window(0)`
Print Window	*obj*<<Print Window	Prints the selected window.
Get Window Size	obj<<Get Window Size	Gets the size of the selected object. Returns an ordered pair showing width and height.
Get Window Position	obj<<Get Window Position	Gets the position of the selected object. Returns an ordered pair.
Report TopReport	*obj*<<Report Report(*obj*)	Returns a display box reference for the report in the platform window. See "Display Box Object References" on page 408 in the "Display Trees" chapter for a discussion. TopReport goes to the top display box, and is useful for BY groups or other cases when several platform reports are in one window.
Journal Window	*obj*<<Journal Window	Appends the contents of the window to the journal.

Options for All Platforms

By

Almost all platforms support the BY argument in the launch command, so By is not documented separately for each platform. See "BY Group Reports" on page 375, for a general discussion.

Title

You can use a Title command across all platforms, which replaces the standard title. For example,

```
Bivariate (X(height), Y(weight), Title("my title"));
```

Axes

For platforms that include graphs, you can customize the axes within the platform script. Options specific to the graphs in each platform are listed.

Automatic Recalculation

If automatic recalc is on, you should use wait(0) commands to let the triggers take effect and do the recalculation.

Additional Notes

Supercategories in Categorical

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 other subset of ratings. Such a group of ratings can be defined in the data through the column property, Supercategories.

```
New Column( "Floss Delimited",
  Character,
  Set Property(
    "Supercategories",
    {Group( "Sleep Related", {"Before Sleep", "Wake"} )}
  ), ...);
```

Supercategories can also be specified in a Categorical launch command, where the properties are listed inside parentheses after the column name:

```
Categorical(
  Supercategories(
```

```
        :Floss Delimited(
           {Group( "Sleep Related", {"Before Sleep", "Wake"} )}
        )
     ), ...);
```

Spline Fits

There are also functions that allow spline fits. In each of the following functions, *x* is a vector of regressor variables, *y* is the vector of response variables, and *lambda* is the smoothing argument. Larger values for *lambda* result in smoother splines.

```
coef = Spline Coef(x, y, lambda);
```

returns a five column matrix of the form

```
knots||a||b||c||d
```

where *knots* is the unique values in *x*, and the spline calculated using the coefficients in the other columns as described with the `Spline Eval` function.

```
yhat=Spline Smooth(x, y, lambda);
```

returns the smoothed predicted values from the spline fit.

```
yhat=Spline Eval(x, coef);
```

evaluates the spline predictions using the `coef` matrix in the same form as returned by `SplineCoef`, in other words, `knots||a||b||c||d`. The *x* argument can be a scalar or a matrix of values to predict. The number of columns of `coef` can be any number greater than 1 and each is used for the next higher power. The powers of *x* are centered at the knot values. For example, the calculation for `coef` of `knots||a||b||c||d` is

j is such that `knots[j]` is the largest knot smaller than *x*

`xx = x-knots[j]` is the centered *x* value

```
result = a[j]+xx*(b[j]+xx*(c[j]+xx*d[j]))
```

or, equivalently,

```
result = a[j] + b[j]*xx + c[j]*xx^2 + d[j]*xx^3
```

Fit Model Effects

Effects in the model can be more than just a list of columns, and have a specialized syntax:

```
Effect( 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. 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
```

To show the launch window and fit the model at the same time, include `Run Model` in the script. To show the launch window but not immediately fit the model, use `Add Script` instead of `Run Model`.

Note: When you select **Analyze > Fit Model** in a data table that has a script named *Model* (or *model*), the launch window is filled in based on the script.

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 this:

```
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 as follows:

- 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 these, 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(...)> )
```

Here is a test script that works on the Dogs data table:

```
manObj = Fit Model(
   Y( logHist0, logHist1, logHist3, logHist5 ),
   Effects( dep1, drug, drug * dep1 ),
   Personality( MANOVA ),
   Run Model
);
manObj << response function( Contrast, Go );
manObj << (response[1] << CustomTest(
      [0 1 0 0,
       0 0 1 0,
       0 0 0 1]
) ); // same as whole model
manObj << (response["contrast"] << (effect["whole model"] << TestDetails));
manObj << (response["contrast"] << (effect[3] << TestDetails));
```

Fit Model Send Command

To send commands to an individual response column's fit, use a syntax like this:

```
fitObj << (responseName << {options, ...});
```

The Send command inside the Send command finds the named response and sends the list of commands to it. If you instead send the options directly to the *fitObj* with a single Send command, the options are sent to all responses.

To send commands to an individual effect, you nest Send commands even further:

```
fitObj << (responseName << ((effectName) << effectOption));
```

DOE Scripting

The DOE platforms (Design of Experiments) are usually used as interactive windows, but are scriptable if you want to drive the platform through scripting. The result of DOE is a data table containing a table property named Model, which is a script to launch Fit Model with the settings appropriate for your design.

A complete list of DOE commands is available in the JMP Scripting Index. Select **Help > Scripting Index**, select **Objects** from the menu, and then search for DOE.

Tuning Commands

There are scripting commands that let you predefine guidelines for the design search by the custom designer. The following commands are described briefly here, and examples also appear in the *Design of Experiments Guide*.

DOE Mixture Sum If you want to keep a component of a mixture constant throughout an experiment then the sum of the other mixture components must be less than one. In defining the factors, you enter the constant mixture component as a constant factor. For example, if you have a mixture factor with a constant value of 0.1, then the command

```
DOE Mixture Sum = 0.9;
```

constrains the remaining mixtures factors to sum to 0.9 instead of the default 1.0.

DOE Starts is the number of random starts used by the custom designer. In some situations, the default number of starts might not produce the design that you want. You can increase the number of starts with the `DOE Starts` command. For example, submitting the JSL statement

```
DOE Starts = 100;
```

overrides the default number of starts and sets the number of starts to 100.

DOE Starting Design For example,

```
DOE Starting Design = matrix;
```

replaces the random starting design with a specified matrix. If a starting design is supplied, the custom designer has only one start using this design.

DOE K Exchange Value By default, the coordinate exchange algorithm considers every row of factor settings for possible replacement in every iteration. Some of these rows might never change. For example

```
DOE K Exchange Value = 3;
```

sets this value to a lower number, three in this case, so the algorithm only considers the most likely three rows for exchange in each iteration.

DOE Bayes Diagonal For example,

```
DOE Bayes Diagonal = vector:
```

This vector is used to modify the diagonal elements of the $X'X$ matrix used for finding the D-optimal design. The supplied vector is added to the current diagonal elements of the $X'X$ matrix.

DOE Sphere Radius constrains the Custom Designer to a sphere instead of a hypercube.

```
DOE Sphere Radius = n
```

where *n* is the radius of the constraining sphere.

Scatterplot Scripting

Use the `Scene3DHardwareAcceleration` command to set the **Use Hardware Acceleration** in Scatterplot 3D. For example,

```
dt = Open( "$SAMPLE_DATA/solubility.jmp" );
Scene3DHardwareAcceleration = 0;
dt << Scatterplot 3D(
   Y( :Name( "1-Octanol" ), :Ether, :Chloroform, :Benzene, :Carbon
   Tetrachloride, :Hexane )
);
```

Process Capability Scripting

Specification Limits in JSL Scripts

Specification limits can be read from JSL scripts, column properties, or from a spec limits data table. As an example of reading in spec limits from JSL, consider the following code example, which places the spec limits inside a `Spec Limits()` expression for the Cities.jmp sample data table:

```
// JSL for reading in spec limits
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 spec limits data table inside an `Import Spec Limits()` expression.

```
// JSL for reading in a spec limits file
Process Capability(
   Process Variables( :OZONE, :CO, :SO2, :NO ),
   Spec Limits(
      Import Spec Limits(
```

```
        "$SAMPLE_DATA/CitySpecLimits.jmp"
)));
```

Control Charts

Customizing Tests in Control Chart Builder

The Customize Tests option lets you design custom tests and select or deselect multiple tests at once. You specify the description, desired number, and label. The option is available only for Variables and Attribute chart types.

```
Control Chart Builder(
  Variables( Subgroup( :DAY ), Y( :DIAMETER ) ),
  Customize Tests( Test 1( 2, "1" ) ), // description, test number, label
  Chart( Position( 1 ),
  Warnings( Test 1( 1 ) ) ), // turn Test 1 on
  Chart( Position( 2 ) )
);
```

Running 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_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 property named QC Alarm Script. To automatically write a message to the log whenever a test fails,

1. Run the script below to save the script as a property to the data table,
2. Run a control chart,
3. Turn on the tests you're interested in. If there are any samples that failed, you'll see a message in the log.

```
CurrentData Table()<<Set Property("QC Alarm Script",
  Write(match(
    QC_Test,1,"One point beyond zone A",
          2,"Nine points in a row in zone C or beyond",
```

```
3,"Six points in a row Steadily increasing or
   decreasing",
4,"Fourteen points in a row alternating up and
   down",
5,"Two out of three points in a row in Zone A or
   beyond",
6,"Four out of five points in a row in Zone B or
   beyond",
7,"Fifteen points in a row in Zone C",
8,"Eight points in a row on both sides of the
   center line with none in Zone C" )));
```

Example 2: Running a chart with spoken tests

```
dt = Open("$SAMPLE_DATA/Quality Control/Coating.jmp");
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`.

Note: Under Windows, in order to have sound alerts you must install the Microsoft Text-to-Speech engine, which is included as an option with the JMP product installation.

Example: $\overline{X}$ - and R-Charts

The following example uses the Coating.jmp sample data (taken from the *ASTM Manual on Presentation of Data and Control Chart Analysis*). The quality characteristic of interest is the Weight column. A subgroup sample of four is chosen.

Run the following script to create the XBar and R charts:

```
Open("$SAMPLE_DATA/Quality Control/Coating.jmp");
Control Chart(Sample Size( :Sample), KSigma(3), Chart Col( :Weight, XBar, R));
```

Sample six indicates that the process is not in statistical control. To check the sample values, click the sample six summary point on either control chart. The corresponding rows highlight in the data table.

Example: $\overline{X}$- and S-charts with Varying Subgroup Sizes

This example uses Coating.jmp, but this time the quality characteristic of interest is the Weight 2 column.

Run the following script to create the XBar and S charts:

```
Open("$SAMPLE_DATA/Quality Control/Coating.jmp");
Control Chart(Sample Size( :Sample), KSigma(3), Chart Col( :Weight 2, XBar,
  S));
```

Weight 2 has several missing values in the data, so you might notice the chart has uneven limits. Although, each sample has the same number of observations, samples 1, 3, 5, and 7 each have a missing value.

Example: Individual Measurement and Moving Range Charts

The Pickles.jmp data in the Quality Control sample data folder contains the acid content for vats of pickles. Because the pickles are sensitive to acidity and produced in large vats, high acidity ruins an entire pickle vat. The acidity in four vats is measured each day at 1, 2, and 3 PM. The data table records day, time, and acidity measurements.

Run the following script to create the charts:

```
Open("$SAMPLE_DATA/Quality Control/Pickles.jmp");
Control Chart(Sample Label( :Date), GroupSize(1), KSigma(3), Chart Col( :Acid,
  Individual Measurement, Moving Range));
```

Example: UWMA Charts

Consider Clips1.jmp. The measure of interest is the gap between the ends of manufactured metal clips. To monitor the process for a change in average gap, subgroup samples of five clips are selected daily. A UWMA chart with a moving average span of three is examined.

Run the following script to create the UMWA chart:

```
Open("$SAMPLE_DATA/Quality Control/Clips1.jmp");
Control Chart(Sample Size(5), KSigma(3), Moving Average Span(3), Chart Col(
  :Gap, UWMA));
```

The point for the first day is the mean of the five subgroup sample values for that day. The plotted point for the second day is the average of subgroup sample means for the first and second days. The points for the remaining days are the average of subsample means for each day and the two previous days.

The average clip gap appears to be decreasing, but no sample point falls outside the 3σ limits.

Example: EWMA Charts

Run the following script to create the EMWA chart from Clips1.jmp:

```
Open("$SAMPLE_DATA/Quality Control/Clips1.jmp");
Control Chart(Sample Size(5), KSigma(3), Weight(0.5), Chart Col( :Gap, EWMA));
```

Example: NP-Charts

The Washers.jmp data in the Quality Control sample data folder contains defect counts of 15 lots of 400 galvanized washers. The washers were inspected for finish defects such as rough galvanization and exposed steel. If a washer contained a finish defect, it was deemed nonconforming or defective. Thus, the defect count represents how many washers were defective for each lot of size 400.

Run the following script to create the *NP*-chart:

```
Open("$SAMPLE_DATA/Quality Control/Washers.jmp");
Control Chart(Sample Size(400), KSigma(3), Chart Col( :Name("# defective"),
  NP));
```

Example: P-Charts

Again, using the Washers.jmp data, we can specify a sample size variable, which would allow for varying sample sizes.

Run the following script to create the *P*-chart:

```
Open("$SAMPLE_DATA/Quality Control/Washers.jmp");
Control Chart(Sample Label( :Lot), Sample Size( :Lot Size), K Sigma(3), Chart
  Col(Name("# defective"), P))
```

Note that although the points on the chart look the same as the *NP*-chart, the *y* axis, Avg and limits are all different since they are now based on proportions.

Example: U-Charts

The Braces.jmp data in the Quality Control sample data folder records the defect count in boxes of automobile support braces. A box of braces is one inspection unit. The number of boxes inspected (per day) is the subgroup sample size, which can vary. The *U*-chart is monitoring the number of brace defects per subgroup sample size. The upper and lower bounds vary according to the number of units inspected.

Run the following script to create the *U*-chart:

```
Open("$SAMPLE_DATA/Quality Control/Braces.jmp");
Control Chart(Sample Label( :Date), Unit Size( :Unit size), K Sigma(3), Chart
  Col( :Name("# defects"), U));
```

Example: C-Charts

C-charts are similar to *U*-charts in that they monitor the number of nonconformities in an entire subgroup, made up of one or more units. *C*-charts can also be used to monitor the average number of defects per inspection unit.

In this example, a clothing manufacturer ships shirts in boxes of ten. Prior to shipment, each shirt is inspected for flaws. Since the manufacturer is interested in the average number of flaws per shirt, the number of flaws found in each box is divided by ten and then recorded.

Run the following script to create the *C*-chart:

```
Open("$SAMPLE_DATA/Quality Control/Shirts.jmp");
Control Chart(Sample Label( :Box), Sample Size( :Box Size), K Sigma(3), Chart
  Col( :Name("# Defects"), C));
```

Phases

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 JSL syntax for setting the phase level limits in control charts is specific. The following example illustrates setting the limits for the different phases of Diameter.jmp:

```
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"))
));
```

Chapter 11

Display Trees

Create and Use Windows

Now comes the chance to do more than program. This chapter shows how to create things in windows and interact with them. These sections discuss different types of display scripting:

- navigating displays to manipulate items in report windows
- creating display trees to build new windows with custom results or custom combinations of standard results

---

Note: You can both create and use some display boxes, but you can only use other display boxes.

To see a list of all display boxes that you can create, select the JMP **Help** menu and then select **Scripting Index > Functions > Display**.

To see a list of all display boxes that can be found in a display tree, select the JMP **Help** menu and then select **Scripting Index > Display Box**. If a display box is in this list but not in **Functions**, then you can send messages to it, but you cannot create it in a script. These display boxes are typically sub-boxes of other display boxes.

For details about scripting 2- and 3-dimensional plots, see the chapters "Scripting Graphs" on page 487 and "Three-Dimensional Scenes" on page 531.

Contents

Manipulating Displays

Reports in JMP are built in a hierarchical manner by nesting and gluing together different types of rectangular boxes. To manipulate reports with scripts, you first need to learn a bit about how these boxes work. You might want to learn more about how this works if:

- You want to manipulate display boxes from existing reports.

  This section introduces JMP reports in general and then discusses how to navigate reports, extract data from them, and change them from JSL.
- You want to construct your own reports.

  See "Constructing Display Trees" on page 423, after reviewing the introduction of this section.

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 script contains invalid display box messages, JMP ignores those errors and continues executing the script.
- When this preference is on, script execution stops when an invalid display box is reached, and an error is sent to the log.

This option can be useful during script development but can cause unwanted log messages for existing scripts.

Introduction to Display Boxes

Here are a few diagrams of the most common types of display boxes in JMP.

Table 11.1 The most common types of display boxes

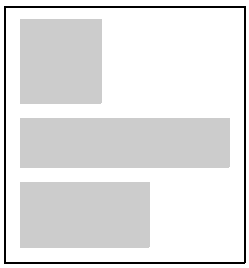
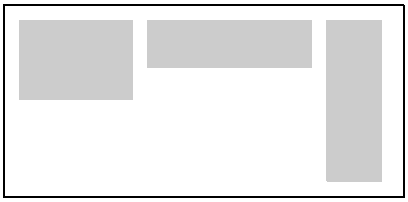
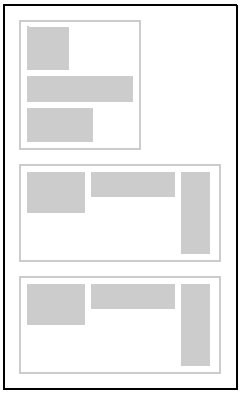
`V List Box` glues boxes together vertically.	

Table 11.1 The most common types of display boxes *(Continued)*

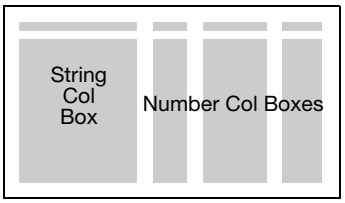
`H List Box` glues boxes together horizontally.



You can nest `V List Box`es and H List Boxes together. Here is a `V List Box` gluing together a `V List Box` and two `H List Box`es.



`Table Box`es are special `H List Box`es whose elements are string and number columns.



`Outline Box`es create an outline hierarchy.

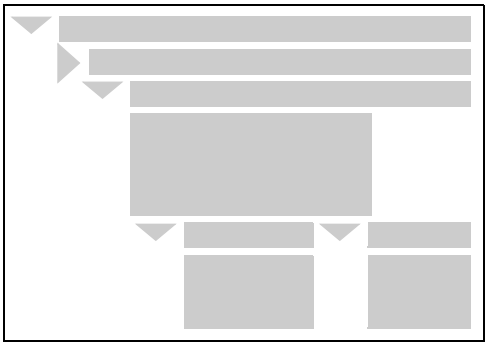


Table 11.1 The most common types of display boxes *(Continued)*

`Picture Box`es glue together axes, frames, and labels to make graphs.	

Note: In JMP, only four containers support multiple children: `H List Box`, `V List Box`, `Outline Box`, and `Panel Box`. Other containers only support one child element.

JSL has some other display box types for custom displays: `Button Box`, `Slider Box`, `Range Slider Box`, `Tab Box` and `Global Box`. These are discussed under "Interactive Display Elements" on page 428. In addition, editable text boxes (`Text Edit Box`) and boxes to make trees similar to those from the Diagram platform (`Hier Box`) are supported.

When a report is displayed, each box asks its children how big they are, so that it can position them and show them in a tidy arrangement.

Here is a report from Bivariate for Big Class.jmp, annotated to show the structure of the display boxes.

Figure 11.1 Display Boxes in a Report

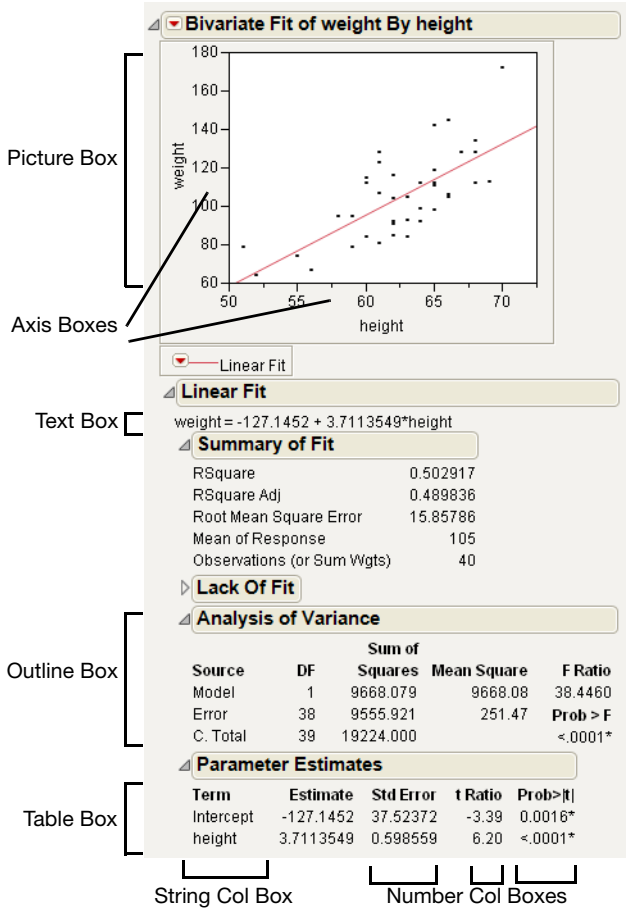


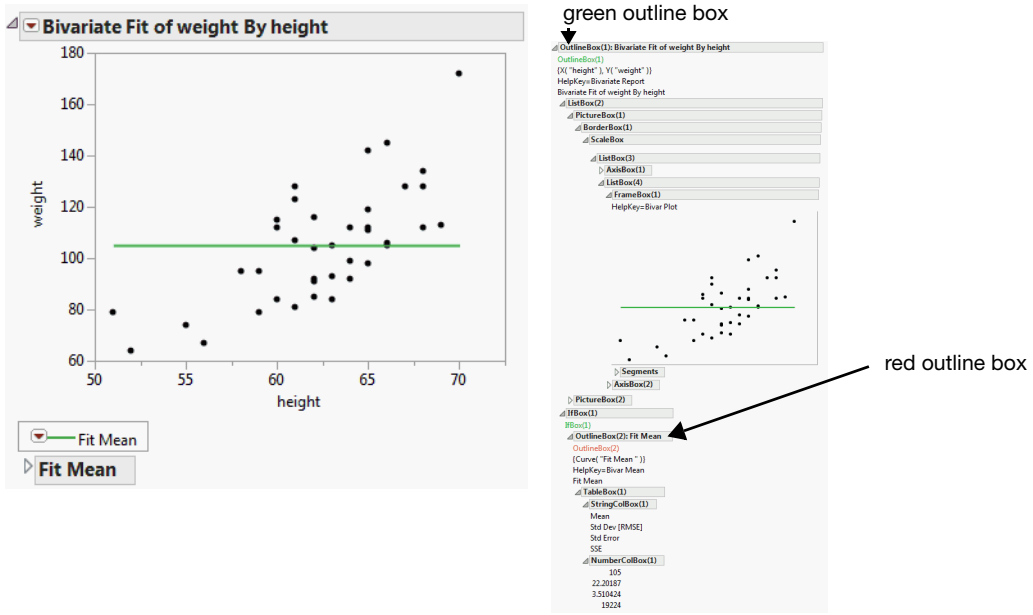
Table 11.5 on page 478, describes the box types and how they arrange their contents. Boxes labeled as "leaf" are those that contain no other boxes inside.

View Display Tree

To examine a report's display box tree, follow these steps:

1. (Optional) Enlarge the report window.
2. Right-click an empty area of the report, and then select **Edit > Show Tree Structure**.

Figure 11.2 Show Tree Structure

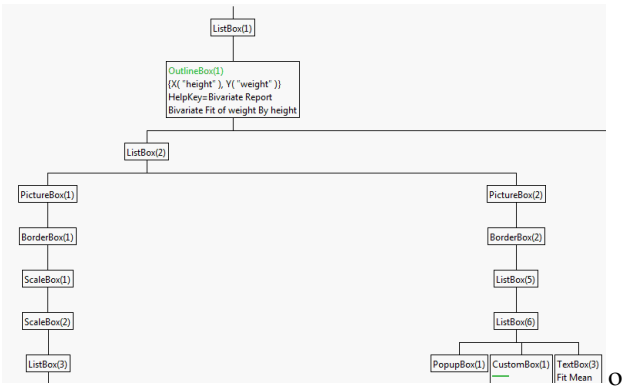


In this example, the red Outline Box represents the closed Fit Mean outline node. The green Outline Box represents the open Bivariate Fit of height By weight outline node. Other boxes in the display tree are shown in their relationship to other boxes.

To view the classic JMP tree structure, hold down **Shift** key and select **Show Tree Structure**.

Place your cursor over the display boxes to outline the hierarchical groups.

Figure 11.3 Partial View of the Classic Tree Structure

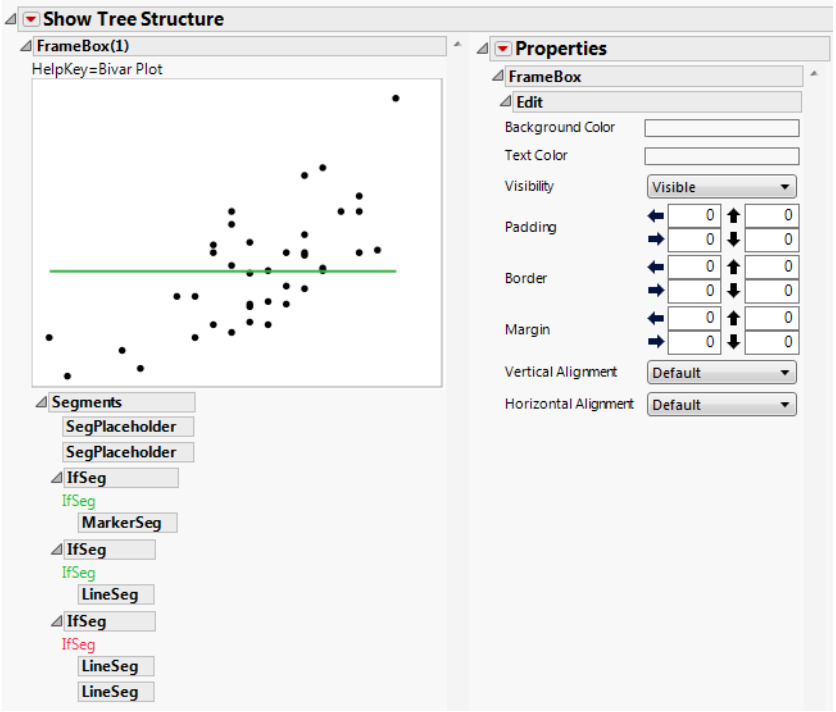


You can also obtain the tree structure through a script: send the `<< Show Tree Structure()` message to any report. Or, send the message to a piece of the report (any display box object) to see the tree structure for just that part of the report.

Show Properties Option

You can edit the properties directly from the display tree. From the red triangle next to Show Tree Structure, select **Show Properties**. Then click on the node that you want to edit. In the example below, the Properties box appears when you click on the plot.

Figure 11.4 Example of Show Properties for Plot



Display Box Object References

A special type of JSL value called a *display box reference* can own or reference a Display Box. (Syntax summaries throughout this manual use *db* as a placeholder for any display box reference, *dt* for a data table reference, and *obj* for a scriptable object reference.)

You can create a display box reference with the `Report` message to access the top of the display tree associated with a scriptable platform.

```
variable = platform object<<report;
```

For example, to obtain a reference to a report for a bivariate analysis, you could say:

```
dt= Open("$SAMPLE_DATA/Big Class.jmp");
biv = bivariate(x(height),y(weight),fit mean, fit polynomial(4));
rbiv = biv<<report;
```

Note: Do not confuse a reference to a *report* with a reference to a *platform*. They are different types of objects and can receive different types of JSL messages. For example, reports can do such things as copy pictures, select display boxes, or close outline nodes. Platforms can do such things as run tests, draw plots, or close entire windows.

By Subscript

If you want to make a reference to another part of the report, the easiest thing to do is use the subscript operator to find it. You can use any of the following methods, which all work by searching for a complete text string. The `"text"` argument can also be any expression that evaluates to a text string. A subscript reference uses the following format:

```
var = db["text"];
```

The line above finds the display box in *db* that has the title *text* and assigns it to the variable `var`. Note that *text* must be the complete title, not just a substring of the title.

Typically you would want to assign the identified part of the report to a variable so that you could send messages to it easily. Such an assignment would have the following format:

```
Variable = rpt ["search string"];
```

For example, to find the Analysis of Variance report for Bivariate, you could say:

```
r1=rbiv["Analysis of Variance"];
```

The sections below describe the various methods for using a subscript operator to reference display boxes.

Wildcard String

You can use a wildcard character (such as “?”) with a substring to match the rest of the title. The following example 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 = Bivariate( Y( :weight ), X( :height ), Fit Mean( {Line Color( {57, 177,
  67} )} ) );
rbiv = biv << report; // returns a reference to the Bivariate report
out2 = rbiv["? Mean"]; // finds "Fit Mean" in the Bivariate report
out2 << Close( 0 ); // opens the Fit Mean outline
```

Box Type and Wildcard String

Another method for performing the same function as the example above, narrows the focus by display box and wildcard search string. For example:

```
out2 = rbiv[Outline Box("? Mean")];
```

The line above finds the Outline Box containing the specified text string in the `rbiv` report and assigns it to the `out2` variable.

Box Type and Index

Another method for locating display boxes uses the *boxType* and an index number, *n*:

```
db[boxType(n)]
```

The line above, finds the *n*th display box of type *boxType*. For example:

```
out2 = rbiv[Outline Box(2)];
```

The line above finds the second Outline Box in the `rbiv` report and assigns it to the `out2` variable.

Tip: To determine the index number of a display box, view the tree structure as described in "Nesting and Precedence" on page 414.

Nested Display Boxes

To reference a display box that appears several layers down a report, use the following format:

```
db[arg1, arg2, arg3, ...]
```

Matches the last argument to a display box that is *contained by* the penultimate argument's outline node, which is in turn contained by the antepenultimate argument, and so on. In other words, it is a way of digging down the generations 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.

```
db[arg1,arg2,arg3]   /* finds the first item, then the next starting after
  that location, and so on */
db[arg1][arg2][arg3] // same as comma version
```

Note that you can string together subscripts of different types.

The code line below would select the third column of the first table of the outline node "Parameter Estimate" under the node "Polynomial Fit Degree=4."

```
rbiv["Polynomial Fit Degree=4"]["Parameter Estimates"][1][3] << select
  << reshow;
```

Display Boxes Without Children

Display boxes with no children are not subscriptable. The following check box has no children, so subscripting makes no sense and produces an error. The last line shows the correct way to ask about individual checks in the column.

```
New Window( "Example",
cb = Check Box( {"One", "Two", "Three"} ) );
cb << Set( 3, 1 ); // selects the third check box
Show( cb[1] << Get( 3 ) ); // wrong
Show( cb << Get( 3 ) ); // correct
   cb << Get(3) = 1; // Log output
```

Note: When numbering `Outline Box`es, it does not matter if the outline is closed or not. However, closed `If Box`es do cause fragments of the tree to disappear entirely.

Wildcards

You can use the wildcard character "?" to represent places in the search string where you want to match any sequence of any characters. This is an important technique for writing general scripts that would work with titles that vary according to the columns analyzed.

For example, you could use a script like this to invoke the Distribution platform for any column starting with "he".

```
dist = Distribution(Y(Column("he?")));
```

You could also use the wildcard character to find a display box by a partial title match as described in "Wildcard String" on page 409.

Sending Messages

The display reference can be used to send scripts to the display elements using the `Send` or << operator. For example, if `out2` is a reference to an outline node, you could ask it to close itself:

```
out2<<close; //closes an outline node
```

`Close` toggles an outline node back and forth between closed and open states, just like clicking the triangle controls in the window. You can include a Boolean argument (1 or 0) for "close or leave closed" or "open or leave open."

```
rbiv["Fit Mean"]<<close;  //toggle
rbiv["Fit Mean"]<<close(1); //close or leave closed
rbiv["Fit Mean"]<<close(0); //open or leave open
```

You can use `Select`, `Reshow`, and `Deselect` to blink the selection highlight on a display box. Notice that 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. For example:

```
dt=Open("$SAMPLE_DATA/Big Class.jmp");
biv = Bivariate( Y( :weight ), X( :height ), Fit Mean( {Line Color( {57, 177,
  67} )} ) );
rbiv = biv <<  report;
out2 = rbiv[Outline Box("? Mean")];
out2 << Close(0);
scbx = rbiv[String Col Box(1)];
for(i=0,i<20,i++,
  wait(.25);
  scbx<<select <<reshow <<deselect <<reshow
  );
```

Learning What You Can Do with a Display Box

To determine the messages that a given display box reference can understand, use Show Properties for the object. For example:

```
dt=Open("$SAMPLE_DATA/Big Class.jmp");
biv = Bivariate( Y( :weight ), X( :height ), Fit Mean( {Line Color( {57, 177,
  67} )} ) );
rbiv = biv <<  report;
Show Properties(rbiv[frame box(1)]);
```

The script above creates a bivariate report and assigns it to variable, rbiv. A list of messages for the specified display box appears in the log window. An example portion of a log appears below:

```
Row Colors [Color](Sets the color of the selected rows.)
Row Markers [Marker](Sets the marker of the selected rows.)
Row Hide and Exclude [Action](Hides and excludes (or unhide or unexcludes) the
  corresponding rows in the data table.)
Row Exclude [Action](Excludes (or unexcludes) the corresponding rows in the
  data table.)
Row Hide [Action](Hides (or unhides) the corresponding rows in the data
  table.)
Row Label [Action](Labels (or unlabels) the corresponding rows in the data
  table.)
Row Legend [Action](Color the rows according to a data column, and insert a
  legend to the right of this frame.)
Row Editor [Action](Bring up the Row Editor window, starting on the first
  selected point.)
Select Matching Cells [Action](Select points that have similar labels to the
  selected rows)
Name Selection in Column [Action](Label the currently selected rows and save
  the value(label) in a column.)
```

```
Select Similar [Action] [Scripting Only]
Get Background Color [Color] [Scripting Only]
Set Background Color [Color] [Scripting Only]
Get Background Fill [Boolean] [Scripting Only]
Set Background Fill [Boolean] [Scripting Only]
Background Color [Color](Change the graph's background color.)
Background Map [Action]
...
```

As you can see, some messages are only available for scripting. Once you know what messages you can use on a display box you can send them to the specified box.

For example, after running the script at the beginning of this section, you send the following lines to the `rbiv` report:

```
fbx=rbiv[frame box(1)];
fbx <<Set Background Color(blue);
```

The background color for the graph's frame box changes to blue. This is the same action as though you right-clicked in the graph and selected **Background Color > Blue**.

Many messages are the same as items in the context-menu for a given object, plus a few more. The next section, "Constructing Display Trees" on page 423, discusses messages for each display box type in more detail.

Note: `Show Properties` also works with data tables and platforms; see "How Can I See All of the Messages that Can be Sent to a Data Table Object?" on page 278 in the "Data Tables" chapter, and "Learning the Messages an Object Responds to" on page 381 in the "Scripting Platforms" chapter.

Using the << Operator

Using the send << operator, messages can be sent to displays as they are being constructed, not just by sending messages to already-constructed displays. This lets you disambiguate between evaluating children arguments and option arguments. It also helps make it clearer which argument is an option, and which is a script to run inside the graph.

For example, before version 5, `H List Box` only accepted boxes in the argument list. Now, commands can be inserted in the list of boxes using the << operator. For example, to insert a journal command in an `H List Box`, do the following:

```
New Window("Title",
  HListBox(
     ...
     <<journal,
     ...
  )
);
```

As another example, the Graph Box constructor accepts the usual named arguments like this:

```
New Window("Title", Graph Box(FrameSize(400,400),XScale(0,25),yScale(0,25),
     <<Background Color("Red")));
```

Alternatively, you can use the send operator to send commands to the Graph Box instead of using the named arguments.

```
New Window("Title",
   Graph Box(
      <<Frame Size(400,400),
      <<XAxis(0,25),
      <<YAxis(0,25),
      <<Background Color("Red")
));
```

Nesting and Precedence

When you stack commands, each command is evaluated left to right.

```
box<<command1<<command2<<command3;
```

The script above sends command1 to box, then command2 to box, then command3 to box. Note that any of the commands can change box before the next command is sent.

```
( (box<<command1) <<command2) <<command3
```

The script above sends command1 to box and gets the result. Command2 is sent to that result, and command3 is sent to the result of command2.

```
x = box<<command1<<command2<<command3;
```

The result of command3 is assigned to the variable x. The first two commands are not assigned to x, although they might have changed box.

```
x = Text Box( "nothing" );
Print( x << Set Text( "the" )
   << Set Text( "first" )
   << Set Text( x << Get Text() || "thing" )
   << Get Text()
);
   "firstthing"
```

If you stack several commands, you might want to use parentheses to group the commands to be sure your script does what you want it to.

Find a Window Quickly

You can find a window that has gotten buried under others by sending a window message to it:

```
rbiv<<zoom window;
```

Customizing Reports

The **Send To Report** and **Dispatch** commands are used in tandem to customize the appearance of a report. For example, they are used to open and close outline nodes, resize graphics frames, or customize the colors in a graphics frame.

To see examples of **Send To Report** and **Dispatch**, run any analysis and change the default appearance of the report. Then, select **Script > Save Script to Script Window** to see the script.

Send To Report contains a list of commands to be sent to the display tree. In the example below, Send To Report contains two Dispatch commands.

Dispatch is used to send a command to a specific part of a display tree. It has four arguments. 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 is the name of a display element, and the third is the display element's type. These two arguments specify which particular part of the display tree is to be sent a command. The command to send is the fourth argument.

In essence, the Dispatch command specifies an outline node in the report (first argument), further specifies something underneath that outline node (second and third arguments), and specifies a command (fourth argument).

For example, open the Big Class sample data set and run the attached **Bivariate** script. This generates a report with a fitted line. Then, open the Lack of Fit outline node, and close the Analysis of Variance outline node. Finally, select **Script > Save Script to Script Window**. The following script appears. (spacing and line breaks are added here for illustration).

```
Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Line( {Line Color( {213, 72, 87} )} ),
   SendToReport(
      Dispatch( {"Linear Fit"}, "Lack Of Fit", OutlineBox, {Close( 0 )} ),
      Dispatch( {"Linear Fit"}, "Analysis of Variance", OutlineBox, {Close( 1
   )} )
   )
);
```

The **Send To Report** command contains two **Dispatch** commands. These correspond to your two customizations to the default report. Examine the first **Dispatch** command in detail.

The first argument says to find an outline node named "Linear Fit". The second and third commands say to further find an Outline Box named "Lack of Fit" underneath the "Linear Fit" outline. The fourth argument is the command 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** commands is to first run a report using the mouse, creating the customizations interactively. Then, examine the script that JMP generates. Remember: the best JSL writer is JMP itself.

Platform Example JSL

Here is a script to build an analysis report from start to finish using JSL. First, open a data table.

```
dt=open("$SAMPLE_DATA/Big Class.JMP");
```

Now launch a bivariate platform and assign it to the platform reference `biv`.

```
biv=bivariate(y(weight),x(height)); // a reference to the PLATFORM
```

To find out what you can do with the platform itself, use `Show Properties` on the platform object:

```
show properties(biv);
   Show Points [Boolean] [Default On]
   Fit Mean [Action](Fits a flat line at the mean.)
   Fit Line [Action](Fits a regression line to the data.)
   Fit Polynomial [ActionChoice] {2,quadratic, 3,cubic, 4,quartic, 5, 6}
   Fit Special [Action](Fitting with transformations on X or Y, or constraints
      on slope or intercept.)
   Fit Spline [ActionChoice] {1000000, stiff, 100000, 10000, 1000, 100, 10, 1,
      0.1, 0.01, flexible, Other…}(Fitting a flexible curve)
   Fit Each Value [Action](Fits a line that goes through the mean Y value of
      each set of unique X values.)
   Fit Orthogonal [ActionChoice] {Univariate Variances, Prin Comp, Equal
      Variances, Fit X to Y, Specified Variance Ratio…}(Fitting where both X and
      Y have error. )
   Density Ellipse [ActionChoice] {0.99, 0.95, 0.90, 0.50, Other…}(The
      bivariate normal contour fitted with the correlation.)
   ...
```

Note that this is a selected portion of the commands available to the Bivariate platform.

The output in the Log window gives a few ideas for messages to send the platform. (For more on this type of scripting, see the chapter "Scripting Platforms" on page 369.)

```
biv<<Fit Spline(1000000)<<Fit Mean;
biv<<show points(0); //hide plotting symbols
biv<<show points(1); //reshow them
biv<<fit polynomial(4, 2); // degree 4 color 2
biv<<fit polynomial(2,4); //degree 2 color 4
```

Next, get the window to a comfortable size and scroll up to the top to see the graph you just tweaked.

```
biv<<size window(500,700);
biv<<scroll window({0,0});
```

Now get to work on the report. First, you need to create a reference, and then see what you can do with it.

```
rbiv=biv<<report; // a reference to the REPORT
show properties(rbiv);
```

The log lists the messages to use with reports (note that this is a selected portion of the commands available to the report):

```
Close [Boolean]
GetOpen [Action] [Scripting Only]
SetOpen [Boolean] [Scripting Only]
Horizontal [Boolean]
GetHorizontal [Action] [Scripting Only]
SetHorizontal [Boolean] [Scripting Only]
Open All Below [Action]
Close All Below [Action]
Open All Like This [Action]
Close All Like This [Action]
Close Where No Outlines [Action]
Get Close Orientation [Action] [Scripting Only]
Outline Close Orientation [Enum] {Auto, Horizontal, Vertical}
Set Title [Action] [Scripting Only]
Get Title [Action] [Scripting Only]
Get Menu Script [Action] [Scripting Only]
Set Menu Script [Action] [Scripting Only]
Set Menu Item State [Action] [Scripting Only]
Get Menu Item State [Action] [Scripting Only]
Get Submenu [Action] [Scripting Only]
Set Submenu [Action] [Scripting Only]
Set Scriptable Object [Action] [Scripting Only]
Get Scriptable Object [Action] [Scripting Only]
Append Item [Subtable]
  Add Text Item [Action]
  Add Outline Item [Action]
  Add Window Reference [Action]
  Add File Reference [Action]
  Add Directory of Files [Action]
  Add All Open Files [Action]
  Add URL Reference [Action]
  Add Script Button [Action]
...
```

Open the Fit Mean node of the outline:

```
rbiv["Fit Mean"]<<close(0);
```

And 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;
//and dig way down in the outline tree:
rbiv["Polynomial Fit Degree=2","Parameter ?", columnbox("Estimate")]<<select;
rbiv<<deselect;
```

To get the second Analysis of Variance item, you would do this:

```
rbiv["Polynomial Fit Degree=2", "Analysis of Variance"]<<select;
```

Now change the format of one of the columns 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 11.5 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

You can even get a single number out of the table. For example, the estimate for the cubic term:

```
terms=pe[Number Col Box("Term")];
for(i=1, i<10 & (terms<<get(i))!="(height-62.55)^3", i++, 0);
estimate = ests<<get(i);
```

0.070394822744608

How does this work? You use a `For`-loop to count down to the row for the term that you want. Recall that 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 `"(height-62.55)^3"` and we have not reached the tenth row," so 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 `Get` on the Estimates column.

This method enables you to use a given result as an argument for further tests. In a process control situation, you might want to keep tabs on a particular result, triggering an e-mail message if it goes outside a certain range.

```
resume = Expr( /* some script to do next iteration */ );
message="Current estimate is " || char(estimate) || " on " || mdyhms(today());
if(estimate<1, resume, //else:
  mail("john.doe@company.com", "estimate exceeded limit", message));
```

You can also get values from boxes as a matrix, which you can then use for further computations or write to a data table. You can also make data tables directly:

```
myMatrix=rbiv[tablebox(4)] << get as matrix;

myVector=rbiv[tablebox(4)][Number Col Box("Sum of Squares")] << get as matrix;
dt<<new column("Sum of Squares",values(myVector));

rbiv[tablebox(4)] << make data table("ANOVA table");
```

Now adjust the scales on the axes.

```
rbiv[axisbox(1)]<<min(70)<<max(170); // adjust Y axis
rbiv[axisbox(2)]<<min(50)<<max(70);  // adjust X axis
```

Continuing with this example, copy the graph at the top of the report. Note that you need to select the picture box containing the graph; selecting just the graph would leave its axes behind.

```
rbiv[PictureBox(1)]<<Copy Picture;
```

You can also save the picture as GIF, PNG (lossless and more compressed than GIFs), JPEG or JPG (good for photographs), and WMF (preserves fonts).

```
rbiv[PictureBox(1)]<<save picture("myGraph.png",png);
rbiv[PictureBox(1)]<<save picture("myGraph.jpeg",jpeg);
rbiv[PictureBox(1)]<<save picture("myGraph.jpg",jpg);
```

Finally, journal the report:

```
rbiv<<journal window;
```

Set Function and Set Script

You can use the `<<Set Script` message to have a display box control (for example, Button Box, Combo Box, Radio Box, etc.) run a script when it is clicked with the mouse. For example:

```
New Window( "Set Script Example",
  ex = Button Box( "Press Me" )
);
ex << Set Script( Print( "Pressed." ) );
```

The script above 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. For example:

```
New Window( "Set Function Example",
   Button Box( "press me",
      <<setFunction(
         Function(
            {this /* functions specified with Set Function get 'this' display
   box */
            },
            this << setButtonName( "thanks" )
         )
      )
   )
);
```

The script above 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”.

Note: You cannot use both a <<Set Script and a <<Set Function at the same time. Use <<Set Function if you need to reference a specific display box object.

Window

Window finds a window by its *"title"* and returns a reference to that window. For example, to close a window titled "Analysis Results" submit the following code:

```
wdw=Window("Analysis Results");
wdw<<closeWindow();
```

- Window() with no arguments returns a list of all windows open in JMP.
- Window(n) returns the *n*th window.
- Window("title") returns the window with the specified title.

Note: If the *n*th window or a window with the specified title does not exist, an empty list is returned instead.

How to Access Built-in Windows

What if you just want your script to open a built-in window? Perhaps you want the user to locate a data table or launch a platform. To present a JMP window, simply omit required arguments from the JSL command for the task. Some examples:

```
dt=Open();          // File > Open window
dt<<Summary();      // Tables > Summary window
myFit=Fit Y by X(); // Analyze > Fit Y by X launch window
myChart=Chart();    // Graph > Chart launch window
```

For modal windows such as `Open()`, further script execution waits until the modal window is dismissed. Whenever JMP encounters a script to launch a platform that lacks column assignments *and* that requires column assignments, the platform's launch window appears, to get column assignments, but continues with the script immediately. That is, JMP does *not* wait for answers, as in the case of `Open()`, and other modal dialog boxes. See the chapter "Scripting Platforms" on page 369, for more about launching platforms.

Using the Pick Windows

You can prompt the user to select a directory using the `Pick Directory` command. 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 for Folder window.

```
path = Pick Directory ("Select a directory.");
```

To let the user select a file, use the `Pick File()` command:

```
path = 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"}
```

For example:

```
{"JMP Files|jmp;jsl;jrn", "All Files|*"}
```

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",
   "$SAMPLE_DATA",
   {"JMP Files|jmp;jsl;jrn",
   "All Files|*"},
```

```
    1, 0, "newJmpFile.jmp");
```

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 nor the default file:

```
Pick File(
  "Select JMP File",
  "",
  {"JMP Files|jmp;jsl;jrn",
  "All Files|*"},
  1, 0, "");
```

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.

If the <Save Flag> is false, the window is an File Open window; if <Save Flag> is true, the window is a Save File window. If <Save Flag> is false, the "Multiple" argument to allow opening multiple files using the one window:

```
Pick File(
  "Select JMP File",
  "",
  {"JMP Files|jmp;jsl;jrn",
  "All Files|*"},
  1, 0, "", "multiple");
```

Note: The buffer size in the computer's physical memory affects the number of files the user can open.

Files in Directory

To obtain a list of filenames in a specific directory, use the Files In Directory command.

```
names = Files In Directory(path, <recursive>);
```

Both filenames and subdirectory names are returned as shown in the following example:

```
names = 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 argument to Files In Directory:

```
names = Files In Directory("$SAMPLE_DATA", recursive);
   { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design Experiment/
      2x3x4 Factorial.jmp", "Design Experiment/Borehole Factors.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("$SAMPLE_DATA", recursive);
   For( i = 1, i <= N Items(names), i++,
      names[i] = Convert File Path("$SAMPLE_DATA") || names[i] );
names;
   {"/C:/Program Files/SAS/JMP/12/Samples/Data/2D Gaussian Process
      Example.jmp",
   "/C:/Program Files/SAS/JMP/12/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 126 in the "Types of Data" chapter for details on working with paths.

Constructing Display Trees

You can use constructor functions to construct your own display and install it in a window. This section shows how to put together displays and send messages to them and concludes with some examples.

Basics

First you need to start a `New Window`, starting with its title, and then you list the items to construct inside the window. All the Display Box constructors end in "`Box`". See "Introduction to Display Boxes" on page 403, to review what each type of display box looks like. There are similar objects that live inside of graphics frames called Display Segs (segments), and these all end in "Seg". We have only made JSL constructors to a small subset of the various Display Boxes and Display Segs in JMP. You can nest display boxes inside each other as needed.

With an assignment, you can make a reference to a new window, so that you can send messages to it. (Throughout this document, *`db`* is a placeholder for a display box reference, *`dt`* for a data table reference, and *`obj`* for a scriptable object reference.)

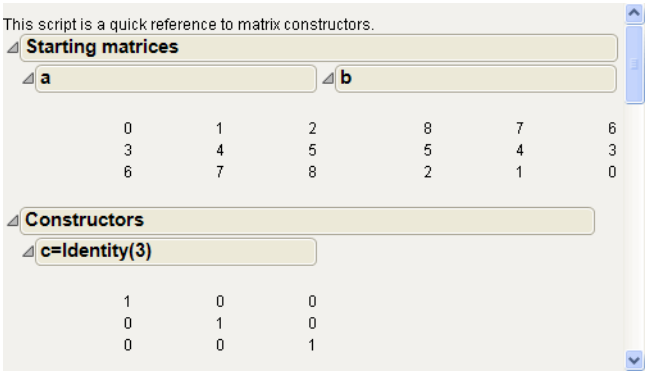
When display objects are created or referred to by JSL, they are freely shared references until they are copied into another display box or until you close the window and they disappear. When you plug a display object into another display tree, JMP makes a copy of it that the new box owns.

Example

This example uses `Outline Box`es, `H List Box`es, and `Matrix Box`es to assemble a partial cheat-sheet for matrix JSL.

```
a=[0 1 2,3 4 5,6 7 8]; b=[8 7 6,5 4 3,2 1 0];
c=identity(3); d=j(3,3,0); e=1::4; f=[+ -,- +];

mcs=New Window("Matrix Cheat Sheet",
   Text Box("This script is a quick reference to matrix constructors."),
      Outline Box("Starting matrices",
      H List Box(
         Outline Box("a",Matrix Box(a)),
         Outline Box("b",Matrix Box(b)))),
   Outline Box("Constructors",
      Outline Box("c=Identity(3)",Matrix Box(c)),
      Outline Box("d=J(3,3,0)",Matrix Box(d)),
      Outline Box("e=1::4",Matrix Box(e)),
      Outline Box("f=[+ -,- +]",Matrix Box(f)),
      Outline Box("Diag(a)", Matrix Box(diag(a))),
      Outline Box("VecDiag(a)", Matrix Box(vecdiag(a))),
      Outline Box("a||b", Matrix Box(a||b )),
      Outline Box("a|\b", Matrix Box(a|/b)),
      Outline Box("a`", Matrix Box(a`))));
```

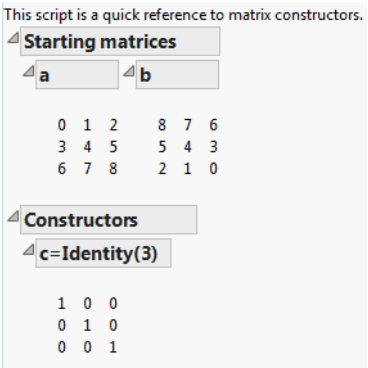
Figure 11.6 A Matrix Example



Now clean up the formatting by sending messages to the display box reference stored in *mcs*:

```
for(i=1,i<12,i++,mcs[matrixbox(i)]<<set format(2,1));
```

Figure 11.7 Formatted Matrix Example



Observe how placing `Matrix Box`es inside `Outline Box`es is a convenient way to arrange items neatly. You could add more topics by adding more outline branches following this pattern.

Updating an Existing Display

Sometimes, you do not know how many display boxes will appear in a future report. For example, you might be writing a generic script that analyzes and reports on one or more variables. You do not know how many display boxes are needed, since 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.

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 made. The interior empty container is given the name `hb`:

```
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<=NItems(effectsList), i++,
 Eval(substitute(
   expr(hb << append(Number Col Box(effectslist[i], varprop[0,i]))),
   expr(i), i)
 );
);
```

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 it 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 script creates a Bivariate report with button box at the top.

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate( Y( height ), X( weight ), Fit Line );
(biv << report)(Outline Box( 1 )) <<
Prepend(
   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:

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate( Y( Height ), X( Weight ), Fit Line );
(biv << report) << Prepend(
   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.

```
x=New Window ("X",
   list = vListBox(
   t1 = Text Box("t1"),
   t2 = Text Box("t2")));
```

```
t1 << Delete;
list << Append(t1 = Text Box ("t1new"));
```

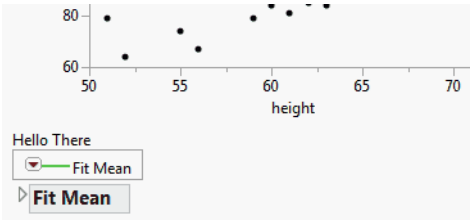
Sib Append

You can use the `Sib Append` message to add a display element to an existing tree. Refer back to the display box tree shown in "View Display Tree" on page 406. Under `ListBox(2)`, you see two picture box trees. `PictureBox(1)` holds the bivariate scatterplot. This is easily determined by seeing the two axis boxes holding the height and weight axes. `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. We want to append a sibling to `Picture Box (1)`, so we send it the `Sib Append` message:

```
Open("$SAMPLE_DATA/Big Class.jmp");
biv=Bivariate( Y( :weight ), X( :height ), Fit Mean( {Line Color( {57, 177,
  67} )} ) );
(biv<<report) [PictureBox(1)] <<SibAppend(Text Box("Hello There"));
```

Figure 11.8 Appending a Sibling Text Box



Updating a Numeric Column

To update a numeric column in a display box table, use the `Set Values` command.

```
Number Col Box<<Set Values ([matrix])
```

The matrix argument specifies the new numbers for the table.

Updating a String Column

To update a string column in a display box table, use the `Set Values` command.

```
String Col Box<<Set Values (<{list}>)
```

The `<{list}>` argument specifies the new strings for the table.

Controlling Text Wrap

Generally, JMP automatically wraps text within a `Text Box`. However, the default wrap point can be overridden with a `Set Wrap` (*n*) message, where *n* is the number of pixels to display before the wrap point.

Bullet Points in Text Boxes

You can also add bullet points using `Bullet Point(1)`:

```
win = New Window( "Bullet List",
   textOne = Text Box( "Lorem ipsum dolor sit amet, consectetur adipisicing
   elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua." )
);
textOne << Bullet Point( 1 );
```

Sending this message to a text box places a bullet in front of the text and indents subsequent lines within that text box.

Interactive Display Elements

JMP has several special types of display boxes that you do not normally see in a JMP platform: `Button Box`, `Slider Box`, `Range Slider Box`, and `Global Box`. These are useful for building custom windows with interactive graphics.

Note: The chapter "Scripting Graphs" on page 487, shows how `Handle` and `MouseTrap` both present controls inside the graph itself.

You can also place buttons, sliders, and edit-field controls outside a graph. Each box type creates separate display boxes that need to be combined in a new window according to the rules described in "Constructing Display Trees" on page 423. For basic effects, you can follow the pattern in the examples shown here and postpone the detailed discussion.

Slider Box

`Slider Box` draws a slider control for picking any value for the variable that you specify, within the range given by the `min` and `max` you specify. Any time the slider is moved, the value given by the current position of the slider is assigned to the global variable. The graph updates accordingly. Thus, `Slider Box` is another way to parameterize a graph.

```
Slider Box(min, max, global variable, script, <set width(n)>, <rescale
   slider(min, max)>);
```

Here is an example:

```
ex = .5;
New Window( "Slider",
```

```
  tb = Text Box( "Value: " || Char( ex ) ),
  sb = Slider Box( 0, 1, ex, tb << Set Text( "Value: " || Char( ex ) ) )
);
```

Range Slider Box

Use `Range Slider Box` to draw a slider that contains two controls for picking a range of values.

```
Range Slider Box(min, max, low_val, high_val, script);
```

Here is a simple example:

```
low = .5;
high = .75;
New Window( "Range Slider",
  tb1 = Text Box( "Low: " || Char( low ) ),
  tb2 = Text Box("High: " || Char(high)),
  sb = Range Slider Box( 0, 1, low, high,
    tb1 << Set Text( "Low: " || Char( low ));
    tb2<< Set Text( "High: " || Char( high ));
  )
);
```

Button Box

`Button Box` draws a button with the name that you specify. Any time the button is clicked, the script executes. The button stays alive and remains available for the duration of the window. You might want to use `Button Box` in combination with sliders or to provide a choice to update a graph to reflect changing data conditions. You can also send a `click()` command to a button object at any time.

```
Button Box("text",<script>);

New Window("Button",
  hello = Button Box("Hello", Print("hello")));
hello<<click();
```

Global Box

`Global Box` shows the name and current value of a JSL global variable. 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. Graphs using the global box automatically update for the new value. If you specify an expression, such as `Sqrt(4)`, the global box first evaluates it and then stores and displays the result, 2.

```
ex = Sqrt(4);
New Window( "Global", Global Box( ex ) );
```

The script above creates a new window named “Global” that displays the following result:

ex=2

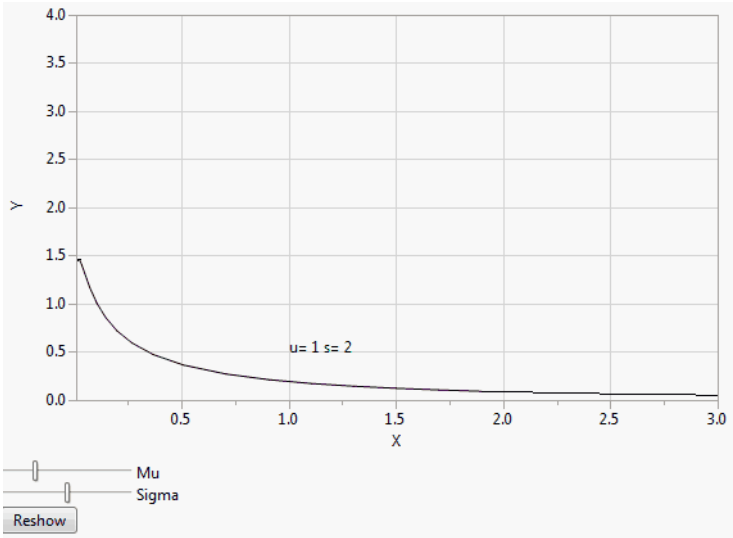
Note: When using `Global Box`, every time the variable is changed it remeasures, reshows, and updates the window which, depending on the number of `Global Box` objects, can affect the refresh rate of the window. JMP recommends avoiding numerous global boxes in deliverable scripts. Global boxes can be replaced with text boxes that the script can manually update when necessary.

Examples

This 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;
New Window("LogNormal Density",
  V List Box(
     gr=Graph Box(Frame Size(500,300),X Scale(0.01,3), Y Scale(0,4),
        Double Buffer, XAxis(Show Major Grid), YAxis(Show Major Grid),
        YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
        text({1,.5},"u= ",lU," s= ",lS)),
     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;
```

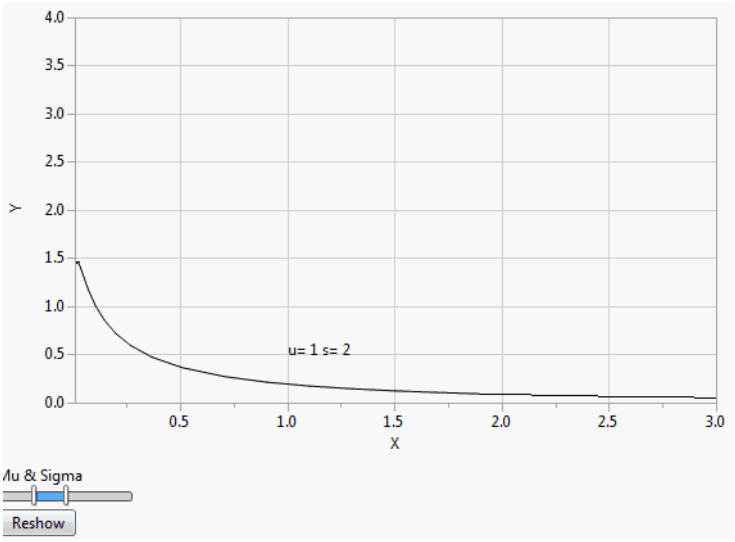
Figure 11.9 Example of Sliders and Buttons in a Report Window



The following script uses a `Range Slider Box()` instead of a `Slider Box()` to perform the same function:

```
// Range Slider LogNormal
lU=1; lS=2;
New Window("LogNormal Density",
  V List Box(
     gr=Graph Box(Frame Size(500,300),X Scale(0.01,3), Y Scale(0,4),
        Double Buffer, XAxis("Show Major Grid"), YAxis("Show Major Grid"),
        YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
        text({1,.5},"u= ",lU," s= ",lS)),
     V List Box(Text Box("Mu & Sigma"),Range Slider Box(0,4,lU,
  lS,gr<<reshow))),
     Button Box("Reshow",gr<<reshow));
show(gr);
gr<<reshow;
```

Figure 11.10 Example of Using Range Slider Box

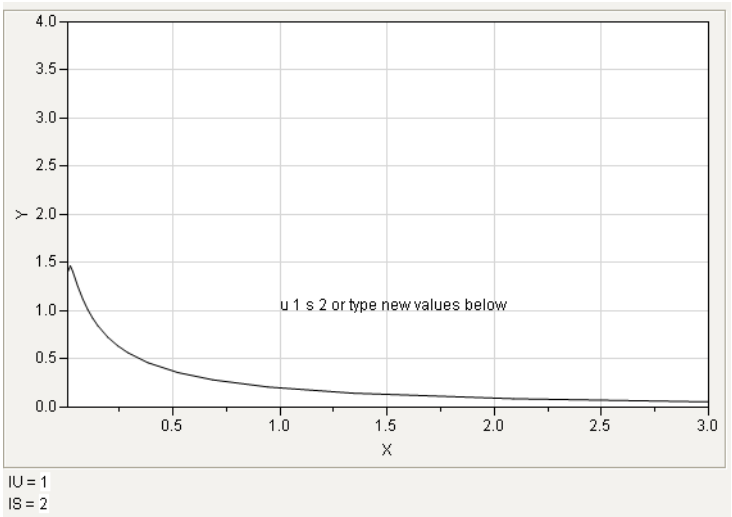


Note: For `Range Slider Box` controls, the upper variable cannot be less that the lower variable and the lower variable cannot exceed the upper variable.

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;
New Window("LogNormal Density",
  V List Box(
     gr=Graph Box(Frame Size(500,300),X Scale(0.01,3), Y Scale(0,4),
        Double Buffer, XAxis("Show Major Grid"), YAxis("Show Major Grid"),
        YFunction(exp(-(log(x)-log(lU))^2/(2*lS^2))/(lS*x*sqrt(2*pi())),x);
        text({1,1},"u= ",lU," s= ",lS," or type new values below")),
     H List Box(Global Box(lU)),
     H List Box(Global Box(lS))));
```

Figure 11.11 Example of Using a Global Box Instead of Sliders



The example under "Drag Functions" on page 525 in the "Scripting Graphs" chapter, shows another use of `Button Box`.

Modal and Non-Modal Windows

Non-modal windows, like the ones used for all JMP launch windows, are identical to reports and other display lists. Each of the following elements can be incorporated into them.

Note: The `Dialog()` function, which creates modal windows, was deprecated in JMP v10 and may not work in future versions of JMP. Use the `New Window()` function with the `Modal` argument in place of the `Dialog()` function.

See "Modal Windows" on page 461 for more details on using `New Window()`. See also the *JSL Syntax Reference* book for syntax details.

The following controls are available for boxes in `New Window()`.

Border Box

```
Border Box (Left(pix), Right(pix), Top(pix), Bottom(pix), Sides(int),
  displaybox)
```

Used to add space around the *displaybox* argument. `Left`, `Right`, `Top`, and `Bottom` add space around the *displaybox* argument. `Sides` draws 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 11.3. To add both an effect and a border, add the two numbers.

For example, this code produces a text box with a border at the top and bottom (Draw Border value of 5) and a white background (Effect value of 32):

```
New Window( "Borders",
   Border Box( Sides( 37 ), Text Box( "Hello World!" ) )
);
```

Table 11.2 Border Box Sides Argument Values

Number	Draw border
0	None
1	Top
2	Left
3	Top and Left
4	Bottom
5	Top and Bottom
6	Left and Bottom
7	Top, Left, and Bottom
8	Right
9	Top and Right
10	Left and Right
11	Top, Left, and Right
12	Bottom and Right
13	Top, Bottom, and Right
14	Left, Bottom, and Right
15	Top, Left, Bottom, and Right

Table 11.3 Border Box Sides Argument Additional Effects

Add to Number Above	Additional Effect
16	Make the border the highlight color as defined in Preferences. The default is blue.

Table 11.3 Border Box Sides Argument Additional Effects *(Continued)*

Add to Number Above	Additional Effect
32	Make the borderbox's background white.
64	Erase the background of the border box's container.

Button Box

```
Button Box("text", <script>)
```

The script above draws a button containing *text*.

You can add a tooltip for your button by sending a Set Tip message to it. For example:

```
Button Box( "Hello", <<Set Tip( "World" ) );
```

The script above creates a button named Hello that, when you hover over it with your mouse, displays a tooltip containing the word World.

Other messages for Button Box include `<<Set Button Name("string")` and `<<Get Button Name`.

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.

You can create a button that contains a menu:

```
New Window( "Test",
   bb = Button Box( "Select a Letter",
      choice = bb << Get Menu Choice;
      Show( choice );
   )
);
bb << Set Menu Items( {"a", "b", "-", "c"} );
```

The `"-"` in this example creates a menu separator. The separator counts as an item in the list. If you select `c` from the menu, `4` is returned, not `3`.

Note: Line-break characters are ignored in button boxes.

Check Box

```
Check Box ({"item 1", "item 2", ...}, <script>)
```

Any number of the items in the check box can be selected simultaneously.

The following example shows how to get the user's selection by sending <<Get Selected to the check box. Each selection is printed to the log.

```
New Window( "Check Box",
   cb = Check Box(
      {"apple", "banana", "orange"},
      scb = cb << Get Selected();
      Show( scb );
   )
);
```

Divide the check box items into multiple columns by dividing the number of items by the number of columns that you want to create. This example also shows how to add buttons that enable the user to select all and clear all check boxes.

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
nameList = Column( 1 ) << Get Values;
hb = H List Box();
c = 1;
For( i = 1, i <= N Items( nameList ), i++,
   Insert Into( tempList, nameList[i] );
   If( c == N Items( nameList ) / 2, // divide the items into 2 columns
      hb << Append( Check Box( tempList ) );
      tempList = {};
      c = 1;
   ,
      c
      ++);
);
selAll = Function( {set},
   {cb},
   cb = hb << child;
   While( !Is Empty( cb ), // while the checkbox is not empty
      cb << Set All( set ); // select all check boxes
      cb = cb << sib; // get the next check box sibling
   );
);
New Window( "Check Box",
   hb,
   Button Box( "Select All", selAll( 1 ) ),
   Button Box( "Clear All", selAll( 0 ) )
);
```

Col List Box

```
Col List Box (<Data Table (<name>), <All>, <width(n)>, <maxSelected(n)>,
  <nlines(n)>, <script>, <MaxItems(n)),<MinItems(n)>, <character | numeric>,
  <onChange(expression)>)
```

`All` specifies that all columns in the current data table should be included. `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.

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:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Get Items Demonstration",
  H List Box(
     chooseme = Col List Box( "All", width( 100 ), nlines( 6 ) ),
     Lineup Box(
        N Col( 1 ),
        Spacing( 3 ),
        Button Box( "Add Column >>",
           listocols << Append( chooseme << GetSelected );
           // Send Get Items to a Col List Box
           Chosen Columns = listocols << GetItems; ),
        Button Box( "<< Remove Column",
           listocols << Remove Selected;
           // Send Get Items to a Col List Box
           Chosen Columns = listocols << GetItems;
        ),
     ),
     // listocols is a Col List Box
     listocols = Col List Box( width( 100 ), nlines( 6 ) ),
  ),
  Text Box( " " ),
  // Show what Get Items returns
  stuff = Global Box( Chosen Columns )
);
```

Col Span Box

Inside a Table Box, create spanned column headers with Col Span Box. The top column header spans two child column headers. One example in JMP is the Confidence Limits header, which spans the Upper and Lower Limit columns below it.

```
Col Span Box( title, children )
```

Combo Box

```
Combo Box({“item 1”, “item 2”, ...}, <script>)
```

Items in a combo box are drawn in a drop-down menu.

The `Editable` argument allows the user to enter text in the combo box. The following script 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.

```
New Window( "Example",
   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. The user must remove focus from the combo box before the `Print()` scripts can run. (Clicking outside the combo box or closing the window removes focus.) On Macintosh, the only way to remove focus is to close the window. An error then occurs because the script cannot run on a deleted combo box.

For cross-platform compatibility, include `!Is Empty` to test for an existing combo box before running the script.

```
New Window( "Example",
   cb = Combo Box(
      {"One", "Two", "Three"},
      Editable,
      If( !Is Empty( cb ),
         selection = cb << GetSelected();
         Print( "Selected: " || selection );
         Print( "Index: " || Char( cb << Get() ) );
      )
   )
);
```

Journal Box

The `Journal Box` function, like other functions ending in `Box`, constructs a Display Box appropriate for gluing together with other Display Boxes to create a display in a window.

The usage is:

```
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 what boxes can be with other boxes, we recommend that the only way that you obtain journal text is to highlight an area, use the Journal command to make a journal containing only that item, save it. Now open the file in a text editor (you might have to change the file extension to do this). 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.

Another way to get journal text is to send <<GetJournal to displayBoxes.

Below is an example that makes a mosaic plot:

```
New Window("MosaicPlot",
TextBox("Here is a mosaic Plot"),
JournalBox("\[ //Note the quoting mechanism here
PictureBox(sub(
BorderBox(top(12), left(5), bottom(5), right(7), sides(143), options(0),
  xmin(0), ymin(0), sub(
ScaleBox(ID(2), axis(scaleType(0), scaleOrig(0), scaleWidth(1),
  widthMajor(0.25),
nbin(4), nminor(0), timeCode(0), ndec(3), ndecSpec(0),
MinInit(0), MaxInit(1), LinearInit, MajorInit(0.25), MinorInit(0),
  NObsInit(0),
options(showMajorTicks, showMinorTicks, showLabels, fixMinimum, fixMaximum)),
  length(180), sub(
ScaleBox(ID(1), length(200), sub(
ListBox(horizontal, near, sub(
ListBox(horizontal, near, sub(
CenterBox(vert, sub(
TextEditBox("age", left))),
AxisBox(side(R), size(33, 180), locked(false), scales(0, 2, 0, 2), ))),
ListBox(vertical, near, sub(
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0),
  ymin(0), sub(
FrameBox(size(200, 180), border(0), flags(0), markerSize(-1), help
  name("Conting Mosaic"), scales(1, 2, 1, 2), seg(
MosaicSeg(
num x(2), num y(6), totals(18, 22),
cross tabs(0.277777, 0.4444444, 0.72222, 0.83333, 0.944444, 1, 0.136366,
  0.31818, 0.63636, 0.863636,
0.909090909090909, 1), sum weight(40), ycolors(5, -2142812212, -2138140980,
  -2134077810, -2134096057, 3), vertical )))))),
NomAxisBox(size(200, 36), sizeID(1, 0), num labels(2), labels(F, M), value(18,
  22), total(40), horizontal, left),
CenterBox(horiz, sub(
TextEditBox("sex")))))),
```

```
NomAxisBox(size(29, 180), sizeID(0, 2), num labels(6), labels("12", "13",
  "14", "15", "16", "17"), value(8, 7, 12, 7, 3, 3), total(40), vertical,
  left),
BorderBox(left(2), bottom(1), right(2), sides(31), options(0), xmin(0),
  ymin(0), sub(
FrameBox(size(10, 180), border(0), flags(0), markerSize(-1), help
  name("Conting Mosaic Single"), scales(0, 2, 0, 2), seg(
MosaicSeg(
num x(1), num y(6), totals(40),
cross tabs(0.2, 0.375, 0.675, 0.85, 0.925, 1),
sum weight(40), ycolors(5, -2142812212, -2138140980, -2134077810,
  -2134096057, 3), vertical)
))))))))))))))
]\") //End of quoting mechanism here
)
```

Line Up Box

```
Line Up Box (NCol(nc), <Spacing(pixels)>, displaybox args)
```

Display boxes specified in the *displaybox* arguments are drawn in *nc* columns. Optional spacing can be specified, in pixels, for the space between columns.

List Box

```
List Box({"item 1", "item 2", ...}, <width(n)>, <max selected(n)>,
  <nlines(n)>, <script>)
```

`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 to display in the box, with a default value of 3.

You can add items to a list box by either using `Append` or `Insert`:

```
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. Use `Clear Selection` to deselect the item.

```
New Window( "Each box clears the other box",
  window:la =List Box({"broccoli", "spinach", "pepper"},
     <<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.

```
New Window( "Example",
   List Box(
      {{"first", "c:\photo.gif"}, {"second","nominal"}},
      width( 200 ),
   )
);
```

Number Col Edit Box

```
Number Col Box("title", numbers)
```

Creates a column named title with numeric entries given in list or matrix form. For example:

```
x = y = z = 0;
New Window( "Example",
   <<Modal,
   Return Result, // extracts values
   Table Box(
      neb =
      Number Col Edit Box(
         "values",
         {x, y, z}
      )
   )
);
   {neb = {2, 4, 6}, Button( 1 )} // results
```

Number Edit Box

```
Number Edit Box(value)
```

Creates an editable number box that initially contains the value argument. For example:

```
New Window( "Example",
   <<Modal,
   neb = Number Edit Box( 5 )
);
x = neb << get;
```

Note that only the **OK** button appears in the window. This is the default behavior for `New Window()`.

Panel Box

```
Panel Box ("title", displaybox args)
```

Encloses the *displaybox* argument in a labeled border.

Popup Box

```
Popup Box({"command1", script1, "command2", script2, ...})
```

Creates a red triangle menu. The following example stores a command list in a variable, and then uses `Popup Box` to display the items.

```
commandList = {
"command1", print("command1"),
"command2", print("command2"),
"command3", print("command3"),
"command4", print("command4"),
"", empty(), //makes a separator line
"command5", print("command5"),
"commandThrow", throw("commandThrow1"),
"commandError", sqrt(1,2,3),
"commandEnd", print("commandEnd")};

New Window("Test Popup",
  Text Box("Popup Test"),
  Popup Box(commandList);
);
```

Figure 11.12 Sample Red Triangle Menu

Note that you can also disable and re-enable the menu using the message `<<enable(Boolean)`. An argument of 1 turns the menu on, and an argument of 0 turns the menu off. Using the previous example, you would assign the popup box to a variable, and then send messages to it:

```
New Window("Test Popup",
  Text Box("Popup Test"),
  mymenu = Popup Box(commandList);
```

```
);

mymenu << enable(0);
```

The red triangle is there, but the menu itself is disabled.

Radio Box

```
Radio Box({"item 1", "item 2", ...}, <script>)
```

Only one item in the radio box can be selected at any time.

Slider Box

```
Slider Box(min, max, global variable, script, <set width(n)>, <rescale
   slider(min, max)>)
```

Draws a slider control for picking any value for the variable, within the minimum and maximum range that you specify. Any time the slider is moved, the value given by the current position of the slider is assigned to the global variable. The graph updates accordingly. Thus, `Slider Box` is another way to parameterize a graph.

```
ex = .5;
New Window( "Example",
   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 );
```

Create Multiple Slider Boxes

To create multiple slider boxes without using an unique global variable for each one:

```
// Create a slider box with only the min and max value
sb1 = Slider Box( 1, 10 );
sb2 = Slider Box( -10, 10 );
// Set the script or the function
sb1 << Set Function( Function({this},{}, Show(this << get, sb2 << get ) ) );
sb2 << Set Script( Show(sb2 << get, sb1 << get) );
// Place the slider boxes in a window
New Window( "title", sb1, sb2 );
```

String Col Edit Box

```
String Col Edit Box("title", {strings})
```

Creates a column named title in a table containing the string items listed. The string boxes are editable. For example:

```
a = b = c = "";
```

```
New Window( "Example",
   <<Modal,
   Return Result,
   Table Box(
      seb =
      String Col Edit Box(
         "names",
         {a, b, c}
         )
      )
);
   {seb = {"do", "re", "me"}, Button( 1 )}
```

Tab Box

```
Tab Box("page title 1", contents of page 1, "page title 2", contents of page
   2, ...)
```

Draws a tabbed window pane.

Tab boxes can also be used in display trees, as in the following example.

```
New Window("test tabbed pages",
   tb = TabBox("First page",
      vlistBox(
         textBox("first line of first page"),
         textBox("second line of first page")
      ),
   "Second page",
      vlistBox(
         textBox("first line of second page"),
         textBox("second line of second page")
      ),
   "Third page",
      vlistBox(
         textBox("first line of third page"),
         textBox("second line of third page")
      )
));
```

Figure 11.13 Tab Boxes

You can specify which tab should be selected by sending `<<SetSelected(n)`, 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:

- `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 script, the first tab title is fairly long.

```
New Window("test tabbed pages",
   tb = TabBox("First page of my tab box",
      vlistBox(
         textBox("first line of first page"),
         textBox("second line of first page")
      ),
   "Second page",
      vlistBox(
         textBox("first line of second page"),
         textBox("second line of second page")
      ),
   "Third page",
      vlistBox(
         textBox("first line of third page"),
         textBox("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)

Tip: `Set Style()` 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.

Text Box

```
Text Box("text")
```

Draws a non-editable text box. Text Boxes are frequently used as labels of other controls.

You can format the text with HTML tags. For example, the following script formats text in bold.

```
w = New Window("Formatted Text",
   Text Box("This is <b>bold</b> text.",
   <<Markup) );
```

Make sure you close nested tags correctly as shown here:

```
"This is <b><i><u>bold italic</u></i></b> text."
```

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 ("text")
```

Draws an editable text box.

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.

For example, the following script sends a message to the log each time the text edit box is changed.

```
New Window("Text Edit Box",
TextEditBox("Change Me", <<Script(Print("Changed")))
)
```

By assigning a reference to the Text Edit Box object, its contents can be accessed. The following script echoes the value of the Text Edit box to the log each time it is changed.

```
New Window("Text Edit Box",
teb=TextEditBox("Change Me", <<Script(Print(teb<<Get Text)))
)
```

Placeholder 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 script produces the Text Edit Box shown in Figure 11.15. The `Hint()` function returns `"mm/dd/yyyy"` and formats it as light gray text.

```
New Window("Example",
   TextEditBox("Current Date"),
   TextEditBox("", Hint("mm/dd/yyyy"))
)
```

Figure 11.15 Text Edit Box with Placeholder Text

Passwords

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 enters with an asterisk. For example, the following line creates a text edit box whose value is the string "a", but only an asterisk is shown in the text edit box.

```
q = Text Edit Box( "a", passwordstyle( 1 ), setscript( Print( "changed!" ) ) )
```

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.

You can also send a text edit box a message to either use password style or to stop using password style.

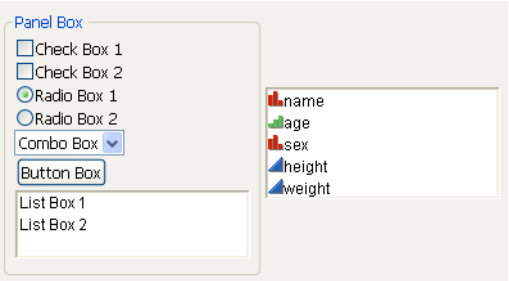
```
q << passwordstyle( 1 ) // set the text edit box to password style
q << passwordstyle( 0 ) // set the text edit box to standard style
```

Complete Example

The following script generates a sample of many controls illustrated above. The Big Class.jmp sample data table is open.

```
New Window("Window Controls",
   Line Up Box (NCol(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



Examples of Getting and Setting Values of Interactive Display Elements

You can use <<Set Selected (Item Number, <State>, <Run Script(0|1)>) command to pre-select an item. You can use this command separately to a saved display box reference, or you can specify it inline as a list box << argument.

To retrieve the selected value, use <<Get Selected, which returns the value of the selected item. << Get Selected Indices returns the index number of the selected item.

```
antennaList = {"Dish","Helical","Polarizing","Radiant Array"};

//method 1: display box reference
New Window("Test List",
 listObj = List Box (antennaList, Print("iList", listObj<<Get Selected,
   listObj<<Get Selected Indices))
);
listObj<<Set Selected(2, 1);

//method 2: inline
New Window("Test List",
 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 you 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"};
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 if you leave out `Run Script()`. However, on `List Box()`, the script runs by default, which is consistent with previous behavior.

Advanced Example

The following example code creates a simplified replica of the Cluster platform launch window, which then launches the platform with the given arguments.

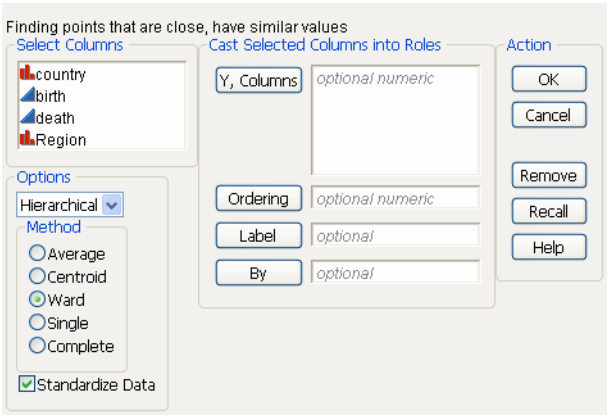
Note: Some functions (`Recall` and `Help`) are not implemented in the script, so an alert window is shown when they are clicked. In addition, switching from **Hierarchical** to **K-Means** clustering does not change anything unlike the real Cluster launch window.

```
// Launch Window for Cluster Platform
dt = Open( "$SAMPLE_DATA/Birth Death.JMP" );
nc = ncol(dt);
lbWidth = 130;
methodList = {"Average","Centroid","Ward","Single","Complete"};
notImplemented = expr(New Window("Feature Not Implemented Yet",<<Modal,
  ButtonBox("OK")));

clusterDlg = New Window("Clustering",<<Modal,
  BorderBox(left(3),top(2),
  VListBox(
     TextBox("Finding points that are close, have similar values"),
     HlistBox(
     VListBox(
        PanelBox("Select Columns",
           colListData=ColListBox(All,width(lbWidth),nLines(min(nc,10)))),
           PanelBox("Options",VListBox(
              comboObj=comboBox({"Hierarchical","K-Means"},<<Set(1)),
              PanelBox("Method",
                 methodObj=RadioBox(methodList,<<Set(3))
           ),
           checkObj=check box({"Standardize Data"},<<Set(1,1))
        )
```

```
      )
   ),
   PanelBox("Cast Selected Columns into Roles",
      LineupBox(NCol(2),Spacing(3),
         ButtonBox("Y, Columns",
            colListY<<Append(colListData<<GetSelected)),
            colListY = ColListBox(width(lbWidth),nLines(5),"numeric"),
         ButtonBox("Ordering",
            colListO<<Append(colListData<<GetSelected)),
            colListO = ColListBox(width(lbWidth),nLines(1),"numeric"),
         ButtonBox("Label",
            colListL<<Append(colListData<<GetSelected)),
            colListL = ColListBox(width(lbWidth),nLines(1)),
         ButtonBox("By",
            colListB<<Append(colListData<<GetSelected)),
            colListB = ColListBox(width(lbWidth),nLines(1))
         )
      ),
      PanelBox("Action",
         LineupBox(NCol(1),
            ButtonBox("OK",
               if ((comboObj<<Get)==1,
                  HierarchicalCluster(
                     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<<CloseWindow
            ),
            ButtonBox("Cancel",
               clusterDlg<<CloseWindow),
               TextBox(" "),
               ButtonBox("Remove",
                  colListY<<RemoveSelected;
                  colListO<<RemoveSelected;
                  colListL<<RemoveSelected;
                  colListB<<RemoveSelected;
               ),
               ButtonBox("Recall",notImplemented),
               ButtonBox("Help",notImplemented))
```

```
            )
        )
    )
  )
);
```

Figure 11.17 The Cluster Launch Window



Send Messages to Constructed Displays

If you assign a construction 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 graphing section shows how to make an interactive sine wave graph. This example automates the interaction, as it were, by sending messages to the frame box inside the window (note the assignment to *tf*).

```
amplitude = 1; freq = 1; phase = 0;
t = New Window( "Wiggle Wave",
  Graph Box(FrameSize(500,300),X Scale(-5,5),Y Scale(-5,5),Double Buffer,
     Y Function(amplitude*Sine(x/freq+phase),x);
     Handle(phase,amplitude,phase=x;amplitude = y);
     Handle(freq,.5,freq=x);
     Text({3, 4},"amplitude: ",Round(amplitude,4),
        {3, 3.5},"frequency: ",Round(freq,4),
        {3, 3},"phase: ",Round(phase,4))));
tf = t[framebox( 1 )];
For(amplitude=-4,amplitude<4,amplitude+=.1,tf << reshow);
```

Use `For` loops for more complex movement:

```
amplitude = 1; 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(0);
);
```

Build Your Own Displays from Scratch

This script uses the Summarize operator 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));
sr=New Window("Summary Results",
   TableBox(
      stringColBox("Age",a),
      NumberColBox("Count",c),
      NumberColBox("Sum",sumHt),
      NumberColBox("Mean",meanHt),
      NumberColBox("Min",minHt),
      NumberColBox("Max",maxHt)));
```

This produces a window called “Summary Results” containing a table as shown in Figure 11.18.

Figure 11.18 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

You can use the usual commands for display boxes:

```
show properties(sr);
sr<<journal;
```

Construct Display Boxes Containing Platforms

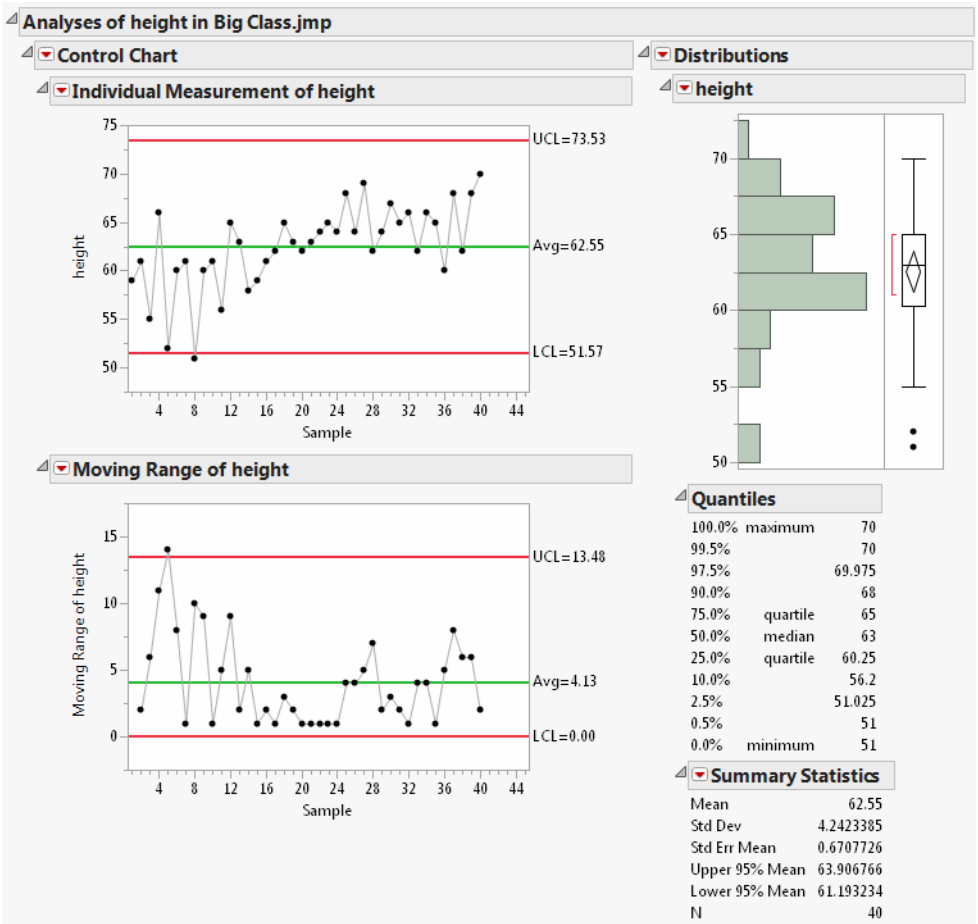
Another type of display that you might want to construct is simply your own combination of results from the analysis platforms in JMP. Simply script the platform inside a display box,

assemble the display boxes into a window, and for ease in routing messages to it later, assign the whole thing to a reference.

This example creates several graphs and reports in one window:

```
dt = Open("$SAMPLE_DATA/Big Class.jmp");
csp=New Window("Analyses of height in Big Class.jmp",
  OutlineBox("Analyses of height in Big Class.jmp",
     HListBox(
        VListBox(cc=Control Chart(chart Col(Height, "Individual
           Measurement", "Moving Range"),K Sigma(3))),
        VListBox(dist=Distribution(columns(Height))))));
```

Figure 11.19 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 this 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;
```

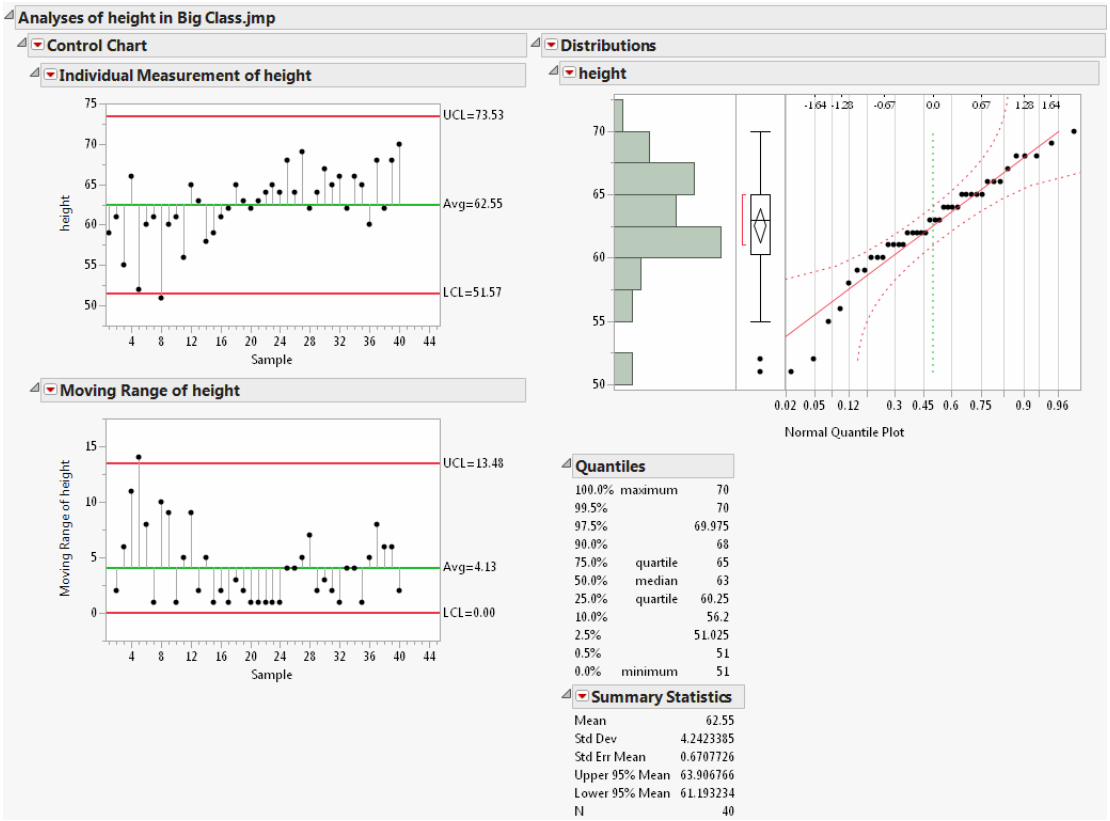
Naturally, you can send messages directly to the platform references themselves. First find out your options:

```
show properties(cc);
show properties(dist);
```

The log shows choices matching the pop-up menus for each platform, as usual. To execute choices from JSL, just direct them as messages to the platform references:

```
cc<<needle; dist<<normal quantile plot;
```

Figure 11.20 Changing a Custom Report



Construct a Custom Platform

An example in "Manipulating expressions" on page 233 in the "Programming Methods" chapter, showed how to use the `SubstituteInto` 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 `SubstituteInto`.

The section "Modal Windows" on page 461 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.

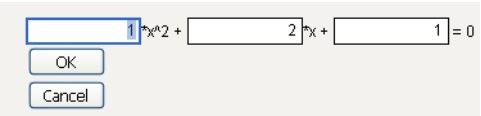
```
//First, open a window to collect coefficients from the user:
myCoeffs = New Window( "Find the roots for the equation",
  <<Modal,
```

```
   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" )
);


/* Second, 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) )};
//Store the solution list:
xx = Eval Expr( x );

/* Third, 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: */
results = Expr(
   xmin = xx[1] - 5;
   xmax = xx[2] + 5;
   ymin = -20;
   ymax = 20;
   myResult = 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(
            framesize( 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} )
         )
      )
   );
);
/* If no (for example, with a=3, b=4, c=5), put up an error window with a
   helpful graph: */
error = Expr(
   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(
            framesize( 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 ) );
//Now it's ready for the test:
If( Is Missing( xx[1] ) | Is Missing( xx[2] ),
   error,
   results
);
```

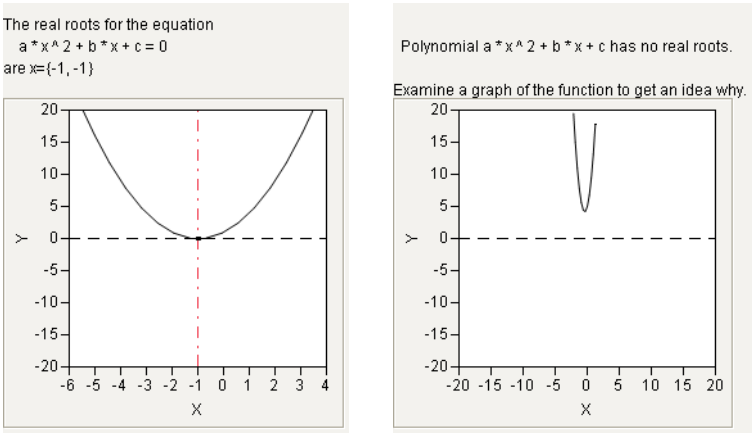
When you run this script, you first see a window like this:

Figure 11.21 Example: A Custom Platform Launch Window



Clicking **OK**, displays a results window (Figure 11.22, 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 11.22, right).

Figure 11.22 Example: The Custom Platform's Report



Sheets

`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. The general approach is to first consider what display boxes that you want, and in what arrangement. Then, create either an `H` or `V Sheet Box` and send it a `Hold` message for each plot. Finally, create interior `H` or `V Sheet Box`es and tell each one which plot it 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.

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Example",
```

Use a `V Sheet Box` to organize the window into two columns.

```
    V Sheet Box(
```

Send it four `Hold` messages, one for each plot. The order matters.

```
<<Hold(Bivariate( // Plot 1
      Y( :weight ),
      X( :height ),
      Fit Line()
)),
<<Hold(Distribution( // Plot 2
      Continuous Distribution(
         Column( :height ),
         Horizontal Layout( 1 ),
         Outlier Box Plot( 0 )
      )
)),
<<Hold(Treemap(Categories( :age ))), // Plot 3
<<Hold(Bubble Plot( // Plot 4
      X( :height ),
      Y( :weight ),
      Sizes( :age ),
      Coloring( :sex ),
      Circle Size( 6.226 ),
      All Labels( 0 )
)),
```

Finally, add two `H Sheet Box`es to the `V Sheet Box` and tell each one which plot it should hold. Each `H Sheet Box` holds two side-by-side plots. They are held by a `V Sheet Box`, so the `H Sheet Box`es are displayed vertically.

```
H Sheet Box(
   Sheet Part("",
```

`Sheet Part` displays a previously defined plot, held by `Excerpt Box`. The first argument is the number of the plot, determined by the order in which you defined the plots. So this first `H Sheet Box` contains the Bivariate plot on the left and the Distribution on the right. `{Picture Box(1)}` designates which picture box from the report to display. Generally speaking, use 1.

```
      Excerpt Box( 1, {Picture Box( 1 )} )
   ),
   Sheet Part("Distribution of height",
      Excerpt Box( 2, {Picture Box( 1 )} )
   )
),
H Sheet Box(
   Sheet Part("",
      Excerpt Box( 3, {Picture Box( 1 )} )
   ),
   Sheet Part("My Title Here",
      Excerpt Box( 4, {Picture Box( 1 )} )
   )
)
```

```
    )
);
```

Finally, a note on the titles for `Sheet Part`. 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". You can define your own title by enter your title as the string. For example, "My Title Here".

Note: For `Sheet Part`, you must include a title, either an empty string or a character string. This argument is mandatory. If you want a blank title, use a string of one or more spaces.

Journals

It is relatively easy to put things into a journal window. It it more difficult to see how to manipulate the journal itself. The following examples show some journal manipulations.

Suppose you start with a report, generated in this case from the Big Class.jmp data table.

```
biv=bivariate (y(weight), x(height));
rbiv=biv << report;
```

To journal the report results, use the `journal window` message.

```
rbiv<<journal window;
```

To save the journal to a file,

```
rbiv<<Save Journal("Macintosh HD:users:username:Documents:test.jrn");
```

To save the journal as HTML,

```
rbiv<<Save HTML("Macintosh HD:users:username:Documents:test.htm");
```

(Note that the above examples use Macintosh standards for filenames. Windows users should adapt the examples to use filename conventions appropriate for your operating system or use the POSIX operating system, which is respected on all platforms.)

To configure the preference for saving journals using GZ compression:

```
rbiv<<Save Journals GZ Compressed(Boolean);
```

Or

```
Preferences(Save Journals GZ Compressed( 1 ));
```

When using a By variable in a script, 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 code creates a Bivariate report with a By group and then journals the entire report:

```
biv=bivariate (y(weight), x(height),by(Sex));
```

```
((report(biv[1]) << parent) << parent) << save journal("test.jrn");
```

Picture Display Type

JSL has a Picture data type, used to store 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 *displaybox*.

```
displaybox<<Get Picture;
```

The function `Expr As Picture` 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, rather than evaluate the expression.

For example,

```
New Window("Formula", Expr As Picture(expr(a+b*c+r/d+exp(x))));
```

Once you have the picture, there are two ways of using it.

1. Incorporate it into a new Display tree, using a *displayBox* constructor.
2. Write it to a file using `Save Picture`.

   ```
   picture<<Save Picture("path", type)
   ```

where *type* can be `WMF`(Windows), `EMF`(Windows), `PICT`(Macintosh), `BMP`(Windows), `JPEG` or `JPG`, or `PNG`.

Modal Windows

JSL supports both modal and non-modal windows.

- Modal windows must be answered immediately. Clicking outside the window produces an error sound. Script execution stops until the user responds to the window.
- Non-modal windows are similar to JMP reports. They do not have to be answered immediately. Platform Launch windows are non-modal.

Note: The `Dialog()` function is deprecated and might not work at all in future versions of JMP. Use either `New Window()` with the `Modal` argument, or `Column Dialog()` for column selection.

Using `New Window()` instead of `Dialog()` means that you can use standard display box constructors, instead of the specialized constructors for `Dialog()`.

Constructing Modal Windows

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. *Modal* means that the user has no choice but to respond to the window. 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.

A few tips for using modal windows in scripts:

1. 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.
2. 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.

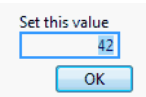
General-Purpose Modal Window

A very simple modal window might request a value for one variable:

```
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. Also notice that a dialog box without at least one `Button` makes no sense, so JMP adds a button if you script a modal window without one.

Figure 11.23 Sample Modal Dialog Box



If you click **OK**, the dialog box closes, it returns {`Button(1)`}, and the script continues. To reference the number set, use `variablebox<<get`.

If you click **Cancel**, the dialog box closes, it returns {`Button(-1)`}, and the script continues. See "Throwing and Catching Exceptions" on page 255 in the "Programming Methods" chapter, for information about canceling a script.

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.

Convert Deprecated Dialog to New Window

`Dialog()` is deprecated and has been replaced with `New Window()` and the `<< Modal` message. Though old scripts based on `Dialog()` still run, we recommend that you use `New Window()` instead.

The sections below illustrate the differences between the `Dialog()` function and the `New Window()` function.

Strings and Lists

A major difference between `Dialog()` and `New Window()` controls is how you specify lists and strings as arguments. In `New Window()`, the items must be placed in a list. In `Dialog()`, items were generally comma-separated. For example, compare the following two instances of a combo box control.

```
// modal New Window
New Window( "Combo Box",
   <<Modal,
   cb = Combo Box( {"True", "False"} ), // Items are in a bracketed list
);

// modal Dialog
Dialog(Title("Combo Box"),
   cb = Combo Box( "True", "False" ), // Items are comma separated
);
```

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, `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 `Dialog()`, you used `H List` and `V List` instead.

The following script creates a window with three horizontal check boxes and an **OK** button:

```
New Window("H List Box",
   <<Modal,
   H List Box(
      kb1 = Check Box( "a"),
      kb2 = Check Box( "b"),
      kb3 = Check Box( "c")
   ),
);
```

Here's the same window created with `Dialog()` and `H List`:

```
Dialog(Title("H List"),
   H List(
         kb1 = Check Box( "a", 0),
         kb2 = Check Box( "b", 0 ),
         kb3 = Check Box( "c", 0)
    ),
);
```

Arranging Items

In `New Window()`, use `Line Up Box` to arrange items in the number of columns that you specify. In `Dialog()`, you used `Line Up`.

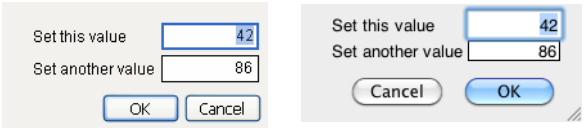
The following script arranges the text boxes in one column and the number edit boxes in another column:

```
New Window("Line Up Box",
   <<Modal,
   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"))

);
```

Clicking **OK** returns the following result in the log window:

```
{Button( 1 )}
```

Figure 11.24 Default Dialog Arrangements for Windows (left) and Macintosh (right)



Here's the same window created with `Dialog()` and `Line Up`:

```
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`, `V List`, and `Line Up` 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 the deprecated `Dialog()` and `New Window()` functions is the usage of `Radio Box`.

In `New Window()`, you must define the `Panel Box` container for `Radio Box`.

```
New Window("Radio Box",
   << Modal,
   Panel Box( "Select",
      rbox = Radio Box( {"a", "b", "c"} )
   )
);
```

In `Dialog()`, the `Radio Buttons` automatically appeared in a panel box.

```
Dialog(Title("Radio Buttons"),
   rb = Radio Buttons( {"a", "b", "c"} ),
);
```

Editable Text Boxes

In `New Window()`, use `Text Edit Box` to create editable boxes that contain specified strings. In `Dialog()`, you use `Edit Text`.

```
New Window("Text Edit Box",
   <<Modal,
   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's the same window created with `Dialog()` and `Edit Text`:

```
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 Box". The window contains a column (labeled "Strings") of editable boxes, each of which contains the specified string:

```
New Window( "String Col Edit Box",
   <<Modal,
   String Col Edit Box(
      "Strings",
      {"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}
   )
);
```

Editable Number Boxes

In `New Window()`, use `Number Edit Box` to create editable boxes that contains the specified numbers. In `Dialog()`, you used `Edit Number`. For example:

```
New Window( "Number Edit Box",
   <<Modal,
   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's the same window created with `Dialog()` and `Edit Number`:

```
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:

```
New Window( "Number Col Edit Box",
   <<Modal,
   nceb = Number Col Edit Box(
      "Random Numbers",
      {num1 = Random Uniform(), num2 = Random Uniform() * 10, num3 =
      Random Uniform() * 100, num4 = Random Uniform() * 1000}
   ),
);
```

Adding Optional Scripts

One of the benefits of using `New Window()` is the ability to add optional scripts to display boxes. In `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:

```
New Window( <<Modal,
   comboObj = Combo Box(
      {"True", "False"},
```

```
        <<Set( 1 ),
        Print( comboObj << Get )
    )
);
```

When the user selects a different value, the selected item number (1 or a 2 in this case, because there are two items in the combo box) is printed to the log.

Comparison of Dialog and New Window

The following two code examples produce identical windows using the deprecated `Dialog()` method and the preferred `New Window()` with the `Modal` option. `New Window` provides more display options and better control over the content and functions of the window.

Dialog Example

```
Dialog(
   Title( "Dialog Example" ),
   H List(
      V List(
         "Radio Frequency Embolism Projection",
         Line up( 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" ) )
   )
);
```

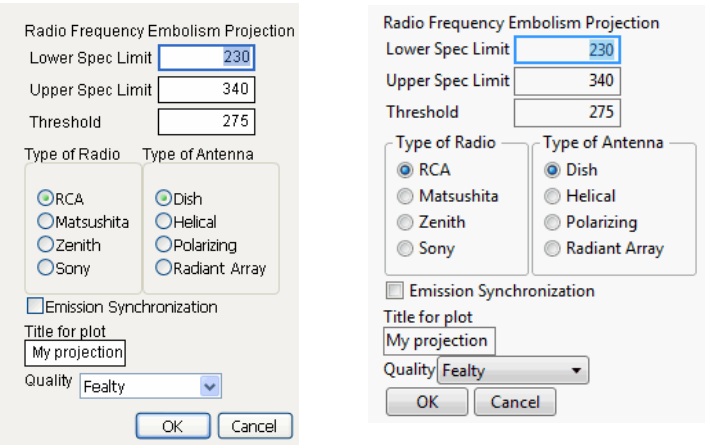
New Window Example

The following example creates the same window with multiple vertical and horizontal list boxes.

```
New Window( "New Window Example",
   <<Modal,
   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 << gettext;
            quality = cb_box2 << get;
         ),
         Button Box( "Cancel" )
      )
   )
);
```

Figure 11.25 Results from Dialog (left) and New Window (right)



Clicking the **OK** button in the `Dialog()` example returns all the variables that you set:

```
{lsl = 230, usl = 340, threshold = 275, type = 1, antenna = 1, synch = 0,
   title ="My projection", quality = 1, Button( 1 )}
```

Clicking the **OK** button in the `New Window()` example returns only the button clicked:

```
{Button( 1 )}
```

Extracting Values

Probably the greatest difference between `Dialog()` and `New Window()` is their difference in extracting values. `Dialog()` allows unloading values after the dialog is closed. For example:

```
dlg = Dialog(
   rb = RadioButtons( {"a", "b", "c"} )
);
Show( dlg["rb"] );  // extracts selection
   dlg["rb"] = 3; // reports "c" selected
```

`New Window()` provides two methods for extracting variable values. By default, you must use individual `<<Get` expressions to retrieve the value for each variable in `New Window()` (see "Extraction Method 2" on page 471). To return the values within `New Window()` in the same manner as `Dialog()`, use the `<<Return Results` option (see "Extraction Method 1" on page 471).

Extraction Method 1

To have a `New Window` script automatically return the results after clicking **OK**, include the `<<Return Result` option after `<<Modal`. For example:

```
nw = New Window("V List Box",
   <<Modal,
   <<Return Result,
   V List Box(
      kb1 = Check Box( "a"),
      kb2 = Check Box( "b"),
      kb3 = Check Box( "c")
   ),
   Button Box("OK")
);
```

If check boxes a and b are selected, the results for the `Return Result` option are:

```
{kb1 = {1}, kb2 = {1}, kb3 = {}, Button( 1 )}
```

Extraction Method 2

To view the output results of the `New Window()` example, add a `<<Get` for each variable. To view the selection, add a `Show` line at the end of the script (without including the `Return Result` option):

```
nw = 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;
      val2=kb2 <<get;
      val3=kb3 <<get;
      ),
);
Show(val1, val2, val3); // returns variables after window closes
```

If check boxes a and b are selected, the results for the script are:

```
val1 = 1;
val2 = 1;
val3 = 0;
```

See "Modal and Non-Modal Windows" on page 433 for additional information on `New Window()` function and its objects. See the *JSL Syntax Reference* for syntax details.

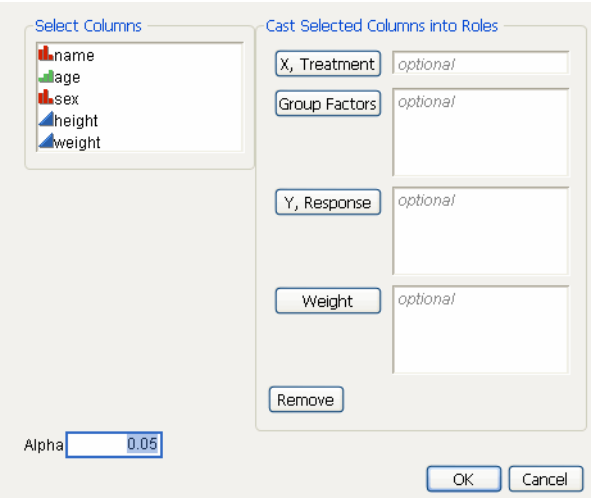
Data Columns

`Column Dialog`, a variant of the deprecated `Dialog` function, lets you prompt for column selections from the current data table or contextual data table, which must already be open.

For example:

```
dt = Open( "$SAMPLE_DATA/Big Class.JMP" );
r = 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 11.26 Column Dialog



This 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*. The resulting list contains the "name" list that is enclosed in parentheses. Lists are *always* returned, although they can sometimes be empty lists. You can include as many as twelve `Col List` clauses.

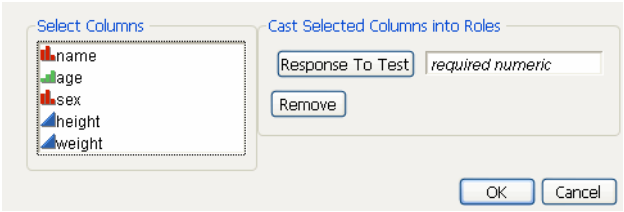
Other items permitted in the deprecated `Dialog` command are permitted in `Column Dialog` also, and have the same functionality. The `OK`, `Cancel`, and `Remove` buttons and the list of columns to choose from are both added automatically.

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.

For example, the following code generates a column dialog box that only allows the selection of exactly one numeric column.

```
rt_Dlg = Column Dialog(
   cv = ColList( "Response To Test", MaxCol( 1 ), MinCol( 1 ), DataType(
   "Numeric" ) )
);
```

Figure 11.27 Restricting Selection of Columns



The `Data Type` choices are `Numeric`, `Character`, and `RowState`.

In addition, use the `Columns` specification to pre-fill some column selections. For example:

```
dlg = Column Dialog(
   xCols = Col List( "X, Factors", Columns( :height ) ),
   yCols = Col List( "Y, Response", Columns( :weight, :age ) )
);
```

assigns height to the X role and weight and age to the Y role.

Unload Results

```
result=New Window("Unload",
   <<Modal,
```

```
    <<Return Result,
    V List Box(
       Line Up Box(NCol(2),
          Text Box("Alpha (0-1)"), a=Number Edit Box(0.05),
          Text Box("Sigma (0-5)"), sd=Number Edit Box(1),
          Text Box("Effect (0-5)"), eff=Number Edit Box(2),
          Text Box("Sample (2-100)"), n=Number Edit Box(2)),
       H List Box(
          Button Box("OK"),
          Button Box("Cancel"))
    )
);
   {a = 0.05, sd = 1, eff = 2, n = 2, Button( 1 )}
```

When you are ready to use some of the values, you have to unload them from the list returned:

```
sd = result["sd"];
eff = result["eff"];
```

Use `EvalList` to evaluate an entire list of assignments all at once:

```
RemoveFrom(result,5); // remove since Button(1) is undefined
EvalList(result);
   {0.05, 1, 2, 2}
```

Constructing Dialogs and Column Dialogs

Table 11.4 shows describes the display box constructors used in deprecated `Dialog()` and `Column Dialog()`. Note the following:

- A Column Dialog automatically includes the list of columns in the data table.
- Both windows automatically include an **OK** button if no buttons are defined.
- A Column Dialog also automatically includes **Cancel** and **Remove** buttons.

Table 11.4 Dialog and Column Dialog Constructors

Constructor	Syntax	Explanation
`Dialog`	`Dialog(` *contents of window* `)`	**Note:** Deprecated in JMP 10. Draws a modal window that includes any of the items listed in this table. The contents must be separated from each other by commas, rather than semicolons.

Table 11.4 Dialog and Column Dialog Constructors *(Continued)*

Constructor	Syntax	Explanation
Column Dialog	Column Dialog(*contents of window*)	Draws a modal window for the user to make column role assignments. The contents must be separated from each other by commas, rather than semicolons.
Col List	*var*=Col List("*role*", <MaxCol(*n*)>, <Datatype(*type*)>)	Available only for Column Dialog(). Creates a selection destination with a *role* button; the user's choice 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 MaxCol(*n*) or MinCol(*n*). You can specify the required data type of the column using Datatype(*type*). The choices for *type* are Numeric, Character, or Rowstate.
List Box	var=List Box({"item", "item", ...}, width(50), max selected(2), nlines(6))	Creates a display box that contains a list of items. List Box in Dialog() allows only one argument (the list). To specify more arguments, include List Box in New Window().
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.
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.
Line Up	Line Up(*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.
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.

Table 11.4 Dialog and Column Dialog Constructors *(Continued)*

Constructor	Syntax	Explanation
`string`	`"string"`	Draws text in the window. For example, can include a labeling string before an Edit Number field. All strings must be quoted.
`Edit Number`	`var=Edit Number( 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. Use `var( Edit Number ( ... )` for `Column Dialog()`. Either syntax works for `Dialog()`.
`Edit Text`	`var=Edit Text( "string", <width(x)> )` `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. Use `var( Edit Text ( ... )` for `Column Dialog()`. Either syntax works for `Dialog()`.
`Radio Buttons`	`var=Radio Buttons( "choice1", "choice2", ... )` `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. Use `var( Radio Buttons( ... )` for `Column Dialog()`. Either syntax works for `Dialog()`.
`Check Box`	`var=Check Box( "Text after box", <1|0> )` `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). Use `var( Check Box( ... )` for `Column Dialog()`. Either syntax works for `Dialog()`.

Table 11.4 Dialog and Column Dialog Constructors *(Continued)*

Constructor	Syntax	Explanation
Combo Box	*var*=Combo Box("*choice1*", "*choice2*", ...) *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. Use var(Combo Box(...) for Column Dialog(). Either syntax works for Dialog().

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 command 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[scriptbox(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` command to get the total number of lines in the script. The `Get Lines` command returns a list of each line in the script as a string.

```
ed << get line count();
ed << get lines();
```

Use the `Reformat` command to automatically format a script for easier reading.

```
ed << reformat();
```

To run an entire script in a script window, use it the `Run` command.

```
ed << run();
```

To close the script window, send the window the `Close Window` command, just like you can do with any JMP window.

```
ww << close window(nosave);
```

Syntax Reference

Table 11.5 contains a summary of common display boxes that appear in reports. The JSL function to create each display box is listed for those that can also be constructed. Many display boxes cannot be constructed through JSL, but are still part of the display tree that can be accessed through JSL.

For information about additional display boxes, see the JMP Scripting Index. (Select **Scripting Index** from the JMP **Help** menu.)

Table 11.5 Display Boxes and Display JSL Functions

Box Type	JSL Function	Description
`Axis Box`		Contains the axis settings such as tick marks, tick labels, axis labels, and label orientation.
`Border Box`	`Border Box(<Left(pixels)>, <Right(pixels)>, <Top(pixels)>, <Bottom(pixels)>, <Sides(0)>, db)`	A container that can be used to add space on one, two, three, or all four sides.
`Button Box`	`Button Box("title", <<Set Icon("path"), script)`	Constructs a button with the *`title`* and executes the *`script`* when the button is clicked.
`Cat Axis Box`		An axis box for a categorical axis.

Table 11.5 Display Boxes and Display JSL Functions *(Continued)*

Box Type	JSL Function	Description
`Cell Plot Box`		Contains a cell plot.
`Center Box`		Centers the display box that it contains.
`Check Box`	`Check Box(list, <script>)`	Constructs a display box to show one or more check boxes.
`Check Box Box`		Contains a single check box within the check box.
`Col List Box`	`Col List Box(<Data Table(<name>)>, <all|character|numeric>, <width(pixels)>, <grouped>, <maxSelected(n)>, <nlines(n)>, <MaxItems(n)>, <MinItems(n)>, <On Change(expr)>, <script>)`	Constructs a display box to show a list box that allows selection of data table columns. Creates a List Box Box to hold the list of columns.
`Combo Box`	`Combo Box(list, <script>)`	Constructs a display box to show a combo box with a menu.
`Context Box`	`Context Box(displayBox, ...)`	Defines a scoped evaluation context. Each Context Box is executed independently of each other. Creates an Eval Context Box.
`Crosstab Box`		Container for a contingency table.
`Display 3D Box`		Container for a three-dimensional `Scene Box`.
`Excerpt Box`	`Excerpt Box(report, subscripts)`	Returns a display box containing the excerpt designated by the report held at number *`report`* and the list of display subscripts *`subscripts`*. The subscripts reflect the current state of the report, after previous excerpts have been removed.
`Expr As Picture`	`Expr As Picture(expr(...), <width(pixels)>)`	Converts `expr()` to a picture as it would appear in the Formula Editor. Creates a `Pict Box`.

Table 11.5 Display Boxes and Display JSL Functions *(Continued)*

Box Type	JSL Function	Description
`Frame Box`		Contains a graphics frame.
`Global Box`	`Global Box(global)`	Constructs a box to display the current value of the *`global`* variable; the value is directly editable, and editing the value automatically forces graphs to update.
`Graph 3D Box`	`Graph 3D Box(properties)`	Constructs a display box with 3-D content. Creates a `Scene Box`.
`Graph Box`	`Graph Box(properties, script)`	Constructs a graph with axes. Might create display boxes such `Axis Box` and `Frame Box`.
`H Center Box`	`H Center Box(display box)`	Returns a display box with the display box argument centered horizontally with respect to all other sibling display boxes. Creates a Center Box.
`H List Box`	`H List Box(display box, ...)`	Creates a display box that contains other display boxes and displays them horizontally. Creates a List Box.
`H Sheet Box`	`H Sheet Box(<<Hold(report), display boxes)`	Returns a display box that arranges the display boxes provided by the arguments in a horizontal layout. The `<<Hold()` message tells the sheet to own the report(s) that are excerpted.
`HierBox`	`Hier Box("text", Hier Box(...), ...)`	Constructs a node of a tree (similar to Diagram output) containing text. `Hier Box` can contain additional `Hier Boxes`, allowing you to create a tree. The *`text`* can be a `Text Edit Box`.
`Icon Box`	`Icon Box("name")`	Constructs a display box containing an icon, where the argument is a name such as Popup, Locked, Labeled, Sub, Excluded, Hidden, Continuous, Nominal, or Ordinal. The argument can also be a path to an image.

Table 11.5 Display Boxes and Display JSL Functions *(Continued)*

Box Type	JSL Function	Description
If Box	If Box(Boolean, display boxes)	Constructs a display box whose contents are conditionally displayed.
Journal Box	Journal Box("Journal Text")	Constructs a display box that displays the quoted string *journal box*. We recommend that you do not generate the journal text by hand.
Lineup Box	Lineup Box(<NCol(n)>, <Spacing(pixels, <vspace>)>, display boxes, ...)	Constructs a display box to show an alignment of boxes in *n* columns.
List Box	List Box({"item", ...}, <width(pixels)>, <maxSelected(n)>, <nLines(n)>, <script>)	Creates a display box to show a list box of selection items. The argument can be a list of two-item lists containing the item name and a string that specifies the modeling type or sorting order. The icon appears next to the corresponding item in the list box.
List Box Box		Created by Col List Box().
Matrix Box	Matrix Box(matrix)	Displays the *matrix* given in the usual array form.
Number Col Box	Number Col Box("title", numbers)	Creates a column named *title* with numeric entries given in list or matrix form.
Number Col Edit Box	Number Col Edit Box("title", numbers)	Creates a column named *title* with numeric entries given in list or matrix form. The numbers can be edited.
Number Edit Box	Number Edit Box(value)	Creates an editable number box that initially contains the *value* argument.
Outline Box	Outline Box("title", display box, ...)	Creates a new outline named *title* containing the listed display boxes.
Page Break Box	Page Break Box()	Creates a display box that forces a page break when the window is printed.

Table 11.5 Display Boxes and Display JSL Functions *(Continued)*

Box Type	JSL Function	Description
Panel Box	Panel Box("title", display box)	Creates a display box labeled with the quoted string *title* that contains the listed display boxes.
Pict Box		Contains a picture object. Created by any function that places a picture into a report window (for example, Picture Box()).
Picture Box	Picture Box(Open(picture), format)	Constructs a box that does nothing except mark the position where pictures are taken when converted to pictures and text, as in outputting to a word processing file.
Plot Col Box	Plot Col Box("title", numbers)	Returns a display box labeled with the quoted string *title* to graph the *numbers*. The numbers can be either a list or a matrix.
Popup Box	Popup Box({"command1", script1, "command2", script2, "",...})	Creates a red triangle menu. The single argument is an expression yielding a list of an even number of items alternating between the command string and the expression that you want evaluated when the command is selected. If the command is an empty string, a separator line is inserted.
Radio Box	Radio Box({"item", ...}, <script>)	Constructs a display box to show a set of radio buttons. The optional script is run every time a radio button is selected.
Range Slider Box	Range Slider Box(min, max, low_val, high_val, script)	Creates and interactive slider control with a low and high variable sliders.
Scene Box	Scene Box(x size, y size)	Creates an *x* by *y*-sized scene box for 3-D graphics.

Table 11.5 Display Boxes and Display JSL Functions *(Continued)*

Box Type	JSL Function	Description
Script Box	Script Box(<script>, <width>, <height>)	Constructs an editable box that contains the quoted string *script*. The editable box is a script window and can both be edited and run as JSL.
Scroll Box	Scroll Box(<size(h,v)>, <flexible(Boolean)>, displayBox, ...)	Creates a display box that positions a larger child box using scroll bars.
Sheet Box		Container for one or more Sheet Panel Boxes.
Sheet Panel Box		Container for a single panel of a sheet layout.
Sheet Part	Sheet Part("title", display box)	Returns a display box containing the *display box* argument with the quoted string *title* as its title.
Slider Box	Slider Box(min, max, global variable, script, <set width(n)>, <rescale slider(min, max)>)	Creates an interactive slider control.
Spacer Box	Spacer Box(<size(h,v)>, <color(color)>)	Creates a display box that can be used to maintain space between other display boxes, or to fill a cell in a LineUp Box.
String Col Box	String Col Box("title", {"string", ...})	Creates a column in the table containing the *string* items listed.
String Col Edit Box	String Col Edit Box("title", {"string", ...})	Creates a column in the table containing the *string* items listed. The string boxes are editable.
Tab Box	Tab Box("page title1", contents of page 1, "page title 2", contents of page 2, ...)	Creates a tabbed window pane. The arguments are an even number of items alternating between the name of a tab page and the contents of the tab page. Creates a Tab List Box.
Table Box	Table Box(display box, ...)	Creates a report table with the *display boxes* listed as columns.

Table 11.5 Display Boxes and Display JSL Functions *(Continued)*

Box Type	JSL Function	Description
`Tab List Box`		Contains `Tab Pane` boxes. Created by `Tab Box`.
`Tab Pane Box`		Contains the display boxes shown on a single tab pane in a tab box.
`Text Box`	`Text Box("text", <arguments>)`	Constructs a box that contains the quoted string *`text`*.
`Text Edit Box`	`Text Edit Box("text", <arguments>)`	Constructs an editable box that contains the quoted string *`text`*. The optional script argument attaches a script to the text box, either by adding the script as a second argument, or by sending the `Set Script` message.
`V Center Box`	`V Center Box(display box)`	Returns a display box with the display box argument centered vertically with respect to all other sibling display boxes. Creates a Center Box.
`V Sheet Box`	`V Sheet Box(<<Hold(report), display boxes)`	Returns a display box that arranges the display boxes provided by the arguments in a vertical layout. The `<<Hold()` message tells the sheet to own the report(s) that is excerpted.
`Web Browser Box`	`Web Browser Box(url)`	Creates a display box that contains a web page. Available only on Windows.

Table 11.6 Subscripts for a Display Box

Symbol	Syntax	Explanation
`[ ]`	*db*`["`*`text`*`"]`	Finds the outline in *db* that has the title *`text`*. Note that *`text`* must be the complete title, not just a substring of the title, but you can use '?' as a wildcard character with a substring to match the rest of the title. For example, `"? Estimates"` to find `"Parameter Estimates"`. See "Display Box Object References" on page 408 for details.
	db`[Outline Box("`*`text`*`")]`	Finds the outline box containing the *`text`*.

Table 11.6 Subscripts for a Display Box *(Continued)*

Symbol	Syntax	Explanation
	db[Column Box("*name*")	Finds the column box containing the *text*.
	db[*boxType*(*n*)]	Finds the *n*th display box of type *boxType*.
	db[*arg1*, *arg2*, *arg3*, ...]	Matches the last argument to a display box that is *contained by* the penultimate argument's outline node, which is in turn contained by the antepenultimate argument, and so on. In other words, it is a way of digging down the generations of an outline tree to identify a nested display box.

Chapter 12

Scripting Graphs

Create and Edit 2-Dimensional Plots

You can run a script inside a graph, which draws inside the graph. You can do this with almost any graph from an analysis platform, or you can create your own new graphs. In both cases, you are storing a JSL script inside a graph, and the script runs each time you display the graph.

For scripting 3-dimensional plots, see the chapter "Three-Dimensional Scenes" on page 531.

Contents

Adding Scripts to Graphs

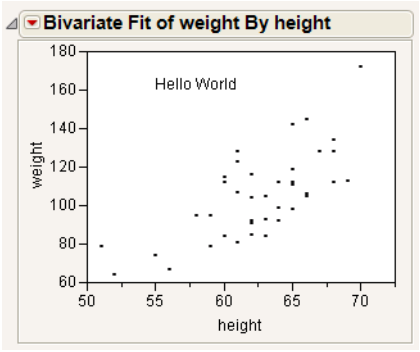
If you context-click (right-mouse-button) on the graphics frame, you can enter or paste JSL commands. The script usually contains drawing commands that run in the context of the graphics frame. For example:

1. Select **Help > Sample Data Library** and open Big Class.jmp.
2. Select **Analyze** > **Fit Y by X**.
3. Choose Height for X and Weight for Y, and click **OK**.
4. Right-click inside the graph.
5. Select **Customize** from the context menu.
6. Click the plus button to add a new graphics script.
7. Type this 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 12.1 Adding a Script to a Graph Interactively



By default, graphics scripts appear on top of the data.

For example, add this line to the graph:

```
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 it to the top. If you want it on top of everything, move it to the bottom. 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 **Script > Save Script to Script Window** from the red triangle menu.

Ordering Graphics Elements Using JSL

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. `Order` can be the keyword `Back` or `Forward` 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` causes the oval to be drawn in the background. `1` 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 behind the points in the plot, and then adds a red oval in front of the blue oval, but still behind the points.

```
dt = Open( "$SAMPLE_DATA/big class.jmp" );

op = dt << Overlay Plot(
  X( :height ),
  Y( :weight ),
  Separate Axes( 1 )
);

Report( op )[Framebox( 1 )] << Add Graphics Script(
  1,
  Description( "Script" ),
  Fill Color( "Blue" );
  Oval( 60, 140, 65, 90, 1 );
);

Report( op )[Framebox( 1 )] << Add Graphics Script(
  2,
  Description( "Script" ),
```

```
   Fill Color( "Red" );
   Oval( 50, 120, 65, 100, 1 );
);
```

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
obj << Paste Frame Contents
obj << Copy Frame Settings
obj << Paste Frame Settings
```

Note the following workaround for copying and pasting histogram bars:

```
dist = Distribution(
   SendToByGroup( {:OPERATOR == "CMB"} ),
   Stack( 1 ),
   Automatic Recalc( 1 ),
   Continuous Distribution(
      Column( :DIAMETER ),
      Horizontal Layout( 1 ),
      Vertical( 0 ),
      Density Axis( 1 ),
      Outlier Box Plot( 0 ),
      CDF Plot( 1 ),
      Test Std Dev( 1 ),
      Test Mean( 0 ),
      Customize Summary Statistics(
         Std Dev( 0 ),
         Std Err Mean( 0 ),
         CV( 1 ),
         Autocorrelation( 1 )
      )
   ),
   By( :OPERATOR ),
   SendToByGroup(
      {:OPERATOR == "CMB"},
      SendToReport(
         Dispatch(
            {"Distributions OPERATOR=CMB", "DIAMETER"},
            "Distrib Histogram",
            FrameBox,
            {Grid Line Order( 2 ), Reference Line Order( 3 ),
            DispatchSeg(
               Hist Seg( 1 ),
               {Line Color( {62, 106, 64} ), Fill Color( "Medium Light Green"
   ),
```

```
            Histogram Color( 36 ), Bin Span( 2, 0 )}
         )}
      ),
      Dispatch(
         {"Distributions OPERATOR=CMB", "DIAMETER"},
         "Quantiles",
         OutlineBox,
         {Close( 1 )}
      ),
      Dispatch(
         {"Distributions OPERATOR=CMB", "DIAMETER"},
         "Summary Statistics",
         OutlineBox,
         {Close( 1 )}
      ),
      Dispatch(
         {"Distributions OPERATOR=CMB", "DIAMETER"},
         "CDF Plot",
         OutlineBox,
         {Close( 1 )}
      )
   )
),
SendToByGroup(
   {:OPERATOR == "DRJ"},
   SendToReport(
      Dispatch(
         {"Distributions OPERATOR=DRJ", "DIAMETER"},
         "Distrib Histogram",
         FrameBox,
         {Transparency( 0.7 ), DispatchSeg(
            Hist Seg( 1 ),
            {Line Color( {112, 112, 56} ), Transparency( 0.7 ),
            Fill Color( {255, 255, 0} ), Histogram Color( -16776960 ), Bin
Span( 3, 0 )}
         )}
      ),
      Dispatch(
         {"Distributions OPERATOR=DRJ", "DIAMETER"},
         "Quantiles",
         OutlineBox,
         {Close( 1 )}
      ),
      Dispatch(
         {"Distributions OPERATOR=DRJ", "DIAMETER"},
         "Summary Statistics",
```

```
            OutlineBox,
            {Close( 1 )}
         ),
         Dispatch(
            {"Distributions OPERATOR=DRJ", "DIAMETER"},
            "CDF Plot",
            OutlineBox,
            {Close( 1 )}
         )
      )
   ),
   SendToByGroup(
      {:OPERATOR == "MKS"},
      SendToReport(
         Dispatch(
            {"Distributions OPERATOR=MKS", "DIAMETER"},
            "Distrib Histogram",
            FrameBox,
            {DispatchSeg(
               Hist Seg( 1 ),
               {Line Color( {56, 56, 112} ), Transparency( 0.7 ), Fill Color(
{0, 0, 255} ),
               Histogram Color( -255 ), Bin Span( 2, 0 )}
            )}
         ),
         Dispatch(
            {"Distributions OPERATOR=MKS", "DIAMETER"},
            "Quantiles",
            OutlineBox,
            {Close( 1 )}
         ),
         Dispatch(
            {"Distributions OPERATOR=MKS", "DIAMETER"},
            "Summary Statistics",
            OutlineBox,
            {Close( 1 )}
         ),
         Dispatch(
            {"Distributions OPERATOR=MKS", "DIAMETER"},
            "CDF Plot",
            OutlineBox,
            {Close( 1 )}
         )
      )
   ),
   SendToByGroup(
```

```
      {:OPERATOR == "RMM"},
      SendToReport(
         Dispatch(
            {"Distributions OPERATOR=RMM", "DIAMETER"},
            "Distrib Histogram",
            FrameBox,
            {DispatchSeg(
               Hist Seg( 1 ),
               {Line Color( {109, 59, 66} ), Transparency( 0.7 ),
               Fill Color( {240, 11, 45} ), Histogram Color( -15731501 ), Bin
   Span( 2, 0 )}
            )}
         ),
         Dispatch(
            {"Distributions OPERATOR=RMM", "DIAMETER"},
            "Quantiles",
            OutlineBox,
            {Close( 1 )}
         ),
         Dispatch(
            {"Distributions OPERATOR=RMM", "DIAMETER"},
            "Summary Statistics",
            OutlineBox,
            {Close( 1 )}
         ),
         Dispatch(
            {"Distributions OPERATOR=RMM", "DIAMETER"},
            "CDF Plot",
            OutlineBox,
            {Close( 1 )}
         )
      )
   )
);

For( i = 2, i <= N Items( dist ), i++,
   Report( dist[i] )[FrameBox( 1 )] << Copy Frame Contents;
   Report( dist[1] )[FrameBox( 1 )] << Paste Frame Contents;
);

New Window( "Test", Outline Box( "Diameter", Report( dist[1] )[Picture Box( 1
   )] ) );
```

Adding a Legend to a Graph

Interactively, you add a legend using the **Row Legend** command. To accomplish this through JSL, send a `Row Legend` message to the display. In this message, specify which column you want to base the legend on, and whether the legend should affect colors and markers.

For example, using Big Class, submit the following JSL to turn on a legend based on the age column, setting both colors and markers by values in the age column.

```
biv = Bivariate( Y( :height), X( :weight) );
Report( biv )[Frame Box(1)] << Row Legend( "age", color(1), marker(1));
```

The `color()` and `marker()` arguments are optional. By default, they mimic the behavior of the window: colors are on by default, and markers are off.

To use a continuous scale if your variable is nominal or ordinal, use the `Continuous Scale(1)` option with the `color(1)` option.

Creating New Graphs From Scratch

Graphics scripts are set up within the `Graph Box` command within a `New Window` command.

```
New Window("window title", <Editable|Dialog>, Graph Box( named arguments,...,
  script));
```

There are two optional keywords after the window name. `Editable` treats the window like a Journal window (so report items can be dragged and dropped onto them). `Dialog` treats the window like a modal window, so it shows up with a gray background on the Macintosh.

`Graph Box` takes named arguments for:

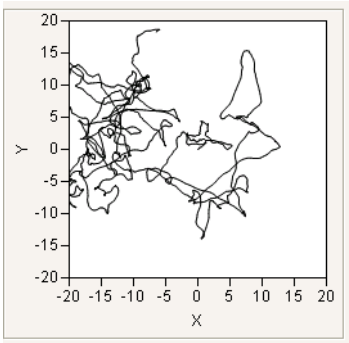
```
FrameSize(horizontal, vertical),          // size in pixels
XScale(xmin, xmax), YScale(ymin, ymax),   // range of x, y axes
X Axis(messages), Y Axis (messages),      // the usual axis controls
Double Buffer                             // for smooth animation
XName, YName                              // names for x, y axes
```

Here is a script that plays a smoothed random walk around the frame. Because this uses random numbers, your display might differ.

```
New Window( "Smoothed Random Walk",
  Graph Box(
     FrameSize( 200, 200 ),
     X Scale( -20, 20 ),
     Y Scale( -20, 20 ),
     x = 0;
     y = 0;
     xi = 0;
```

```
        yi = 0;
        For( i = 0, i < 2000, i++,
           xi = .9 * xi + Random Normal() / 10;
           yi = .9 * yi + Random Normal() / 10;
           xx = x + xi;
           yy = y + yi;
           xx = If(
              xx < -20, -20,
              xx > 20, 20,
              xx
           );
           yy = If(
              yy < -20, -20,
              yy > 20, 20,
              yy
           );
           Line( {x, y}, {xx, yy} );
           x = xx;
           y = yy;
        );
     )
  );
```

Figure 12.2 Creating a Graph



Making Changes to Graphs

You can also make changes to graphs through scripting. For example, create a window with a graph and get a reference to the report:

```
open("$SAMPLE_DATA/Big Class.jmp");
biv = Bivariate( Y( :weight ), X( :height ), Fit Line );
rbiv = biv<<report;
```

Using subscripting, you can send messages to any part of that report. Use **Show Tree Structure** to discover how to reference them. See "View Display Tree" on page 406 in the "Display Trees" chapter for details about the tree structure. See "By Subscript" on page 409 in the "Display Trees" chapter on using subscripts. This line re-sizes the graph to 400 pixels by 400 pixels:

```
rbiv[frame box(1)]<<frame size(400,400);
```

To see a list of possible messages for any given display box object, use `Show Properties`. For example, here is a partial list of messages that you can send to an axis:

```
Show Properties(rbiv[axis box(1)]);
  Axis Settings [Action](Bring up the Axis window to change various settings.)
  Revert Axis [Action](Restore the settings that this axis had originally.)
  Add Axis Label [Action]
  Remove Axis Label [Action]
  Save To Column Property [Action](Save the Axis settings as an Axis property
     in the data column associated with this axis.)
  Set Width [Action] [Scripting Only]
  Axis [Subtable] [Scripting Only]
    Scale [Enum] {Linear, Log, Exp Prob, Weibull Prob, Logistic Prob, Frechet
     Prob, Lognormal Prob, Cube Root}
    Min [Numeric]
    Max [Numeric]
    Inc [Numeric]
    Tick Font [Action]
    Interval [Enum] [Scripting Only]  {Numeric, Year, Month, Week, Day, Hour,
     Minute, Second}
  ...
```

For example, to change the font of both axis labels (weight and height in the example above, which are both text edit boxes attached to the axis boxes) 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( 2 )] << set font( "Arial Black" );
rbiv[Text Edit Box( 2 )] << set font style( "Italic" );
rbiv[Text Edit Box( 2 )] << set font size( 12 );
```

For all display boxes, `<<Set Font Name` messages are ignored if the font is not installed on the computer.

Graphing Elements

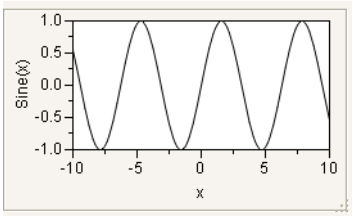
You can use the following commands inside `Graph Box` statements. This chapter focuses on the JSL that is specific to graphing, but you can also use general script commands like `For`, `While`, and so on.

Plotting Functions

A `YFunction` operator is used to draw smooth functions. The first argument is the expression to be plotted, and the second argument is the name of the X variable in the expression.

```
New Window( "Sine Function",
   Graph Box(
      FrameSize( 200, 100 ),
      X Scale( -10, 10 ),
      Y Scale( -1, 1 ),
      xName( "x" ),
      yName( "Sine(x)" ),
      Y Function( Sine( x ), x )
   )
);
```

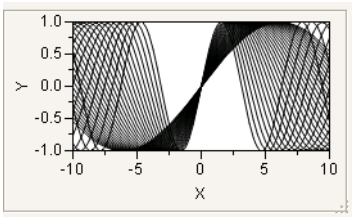
Figure 12.3 Sine Wave



You can use `For` to overlap several sine waves:

```
New Window( "Overlapping Sine Waves",
   Graph Box(
      FrameSize( 200, 100 ),
      X Scale( -10, 10 ),
      Y Scale( -1, 1 ),
      For( i = 1, i <= 4, i += .1,
         Y Function( Sine( x / i ), x )
      )
   )
);
```

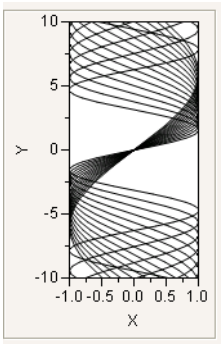
Figure 12.4 Overlapping Sine Waves



Similarly, an `XFunction` is for drawing a plot where the symbol is varied on the Y variable.

```
New Window( "Overlapping Sine Waves",
   Graph Box(
      FrameSize( 100, 200 ),
      X Scale( -1, 1 ),
      Y Scale( -10, 10 ),
      For( i = 1, i <= 4, i += .2,
         X Function( Sine( y / i ), y )
      )
   )
);
```

Figure 12.5 Overlapping Sine Waves Along the X-Axis

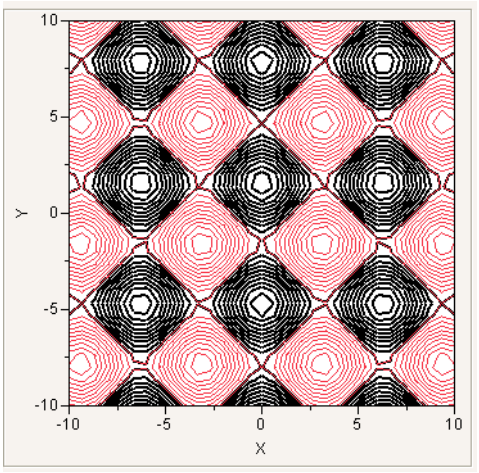


`ContourFunction` is an analogous way to represent a three-dimensional function in a two-dimensional space. The final argument specifies the value(s) for the contour line(s), and it can be a value, an indexed range of values using `::`, or a matrix of values.

```
New Window( "Bird's eye view of the egg-carton function",
   Graph Box(
      FrameSize( 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 12.6 Egg Carton Function



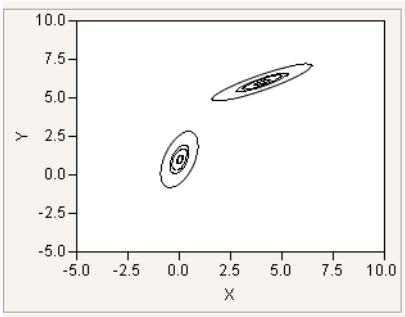
`Normal Contour` draws normal probability contours for *k* populations and two variables. The first argument is a scalar probability or a matrix of probability values for the contours, and subsequent arguments are matrices to specify means, standard deviations, and correlations. The mean and standard 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*. For 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.

```
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 12.7 Normal Contour Function



`Normal Contour` is thus a generalized way to accomplish effects like Bivariate's density ellipses, which are demonstrated to good effect with the Football sample data (just open the data table Football and run its stored Bivariate script).

Gradient Function

The syntax of the gradient function is

```
Gradient Function(expression, xname, yname, [zlow, zhigh], ZColor([colorLow,
  colorHigh]), <XGrid(min, max, incr)>, <YGrid(min, max, incr)>)
```

This 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. To implement it, use the following syntax.

`GradientFunction(`	
`expression`	the expression to be contoured, which is a function in terms of the two variables that follow

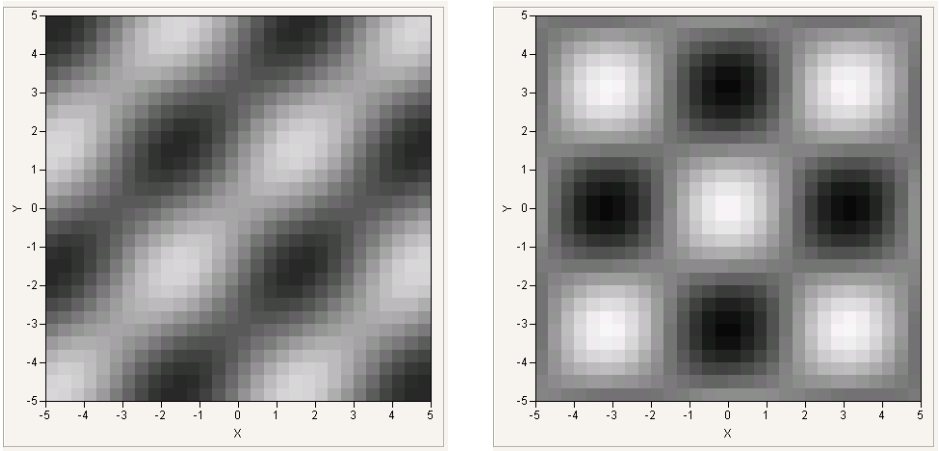
`xname,yname,`	the two variable names used in the expression
`[zlow, zhigh],`	the low and high expression values the gradient is scaled between
`ZColor([colorLow, colorHigh])`	the colors corresponding to the low value and the high value
`<XGrid(min, max, incr),>` `<YGrid(min, max, incr)>` `)`	optional specification for the grid of values

The `ZColor` values must be numeric codes, rather than names. You can use the color menu indices (0=black, 1=grey, 2=white, 3=red, 4=green, 5=blue, and so on) found in "Colors" on page 516.

The following example script uses the `Gradient` function, with the picture showing two frames of the animation.

```
phase = 0.7;
New Window( "Gradient Function",
   a = Graph(
      FrameSize( 400, 400 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      "DoubleBuffer",
      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 12.8 Gradient Function



Getting the Properties of a Graphics Frame

There are several functions that 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 distance from the left to right edge of the displaybox.

X Range Returns the *x*-value for the left edge of the graphics frame.

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.

Examples:

The first line calculates the right edge, and the second line calculates the top edge.

```
rightEdge = X Origin() + X Range();
topEdge = Y Origin() + Y Range();
```

Adding a Legend

You can add a legend to a graph, using the **Row Legend** command. The following example uses the Fitness.jmp sample data file, and sets colors and markers based on the Age column, and adds a legend to the plot.

```
biv = Bivariate(Y(:Oxy), X(:Runtime)); //generate a scatterplot
Report(biv)[Frame Box(1)] << Row Legend (:Age, color(1), marker(1));
```

The `marker()`argument has binary arguments. If you specify 1, the markers are set to each point.

You can specify the following settings for colors:

- `Color(0)` turns the color legend on.
- `Color(1)` turns the color legend off.

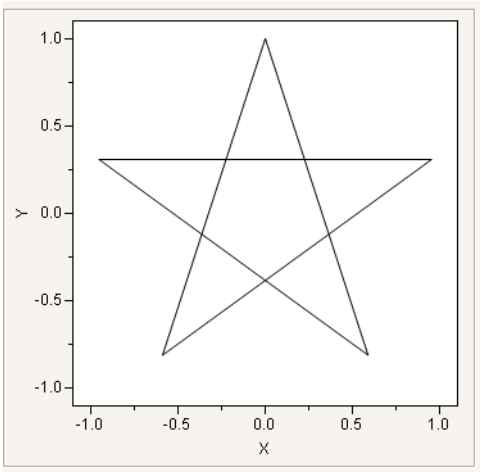
Drawing Lines, Arrows, Points, and Shapes

Lines

`Line` draws lines between points.

```
New Window( "Five-point star",
   Graph Box(
      framesize( 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 12.9 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, so you can use either row or column vectors, as long as you have the same number of elements in each matrix. The following would be equivalent:

```
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
```

Thus, the star example could also be drawn this way. Note that it must use full `Matrix({...})` notation rather than `[ ]` shorthand since the entries are expressions.

```
new window("Five-point star",
  graph box(framesize(300, 300), xscale(-1.1, 1.1), yscale(-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 you specify. Similarly `VLine` draws a vertical line down the graph at the *x*-value you specify. Both commands support drawing multiple lines by using a matrix of values in the *y* argument. These are illustrated in the example under "MouseTrap" on page 524.

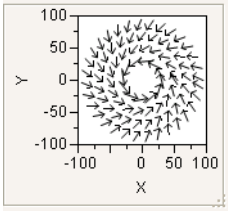
Arrows

Similarly, `Arrow` draws an arrow from the first point to the second. 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.

For example, the following script draws arrowheads with the default length.

```
New Window( "Hurricane",
   Graph Box(
      FrameSize( 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 );
            Arrow( {x, y}, {x2, y2} );
         );
      )
   )
);
```

Figure 12.10 Drawing Arrows

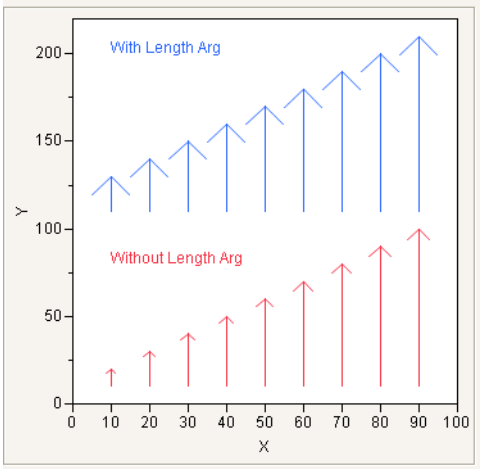


This script compares drawing with a specified length (19 pixels) and drawing with the default arrow head size.

```
New Window( "Arrow Heads", 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 12.11 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.

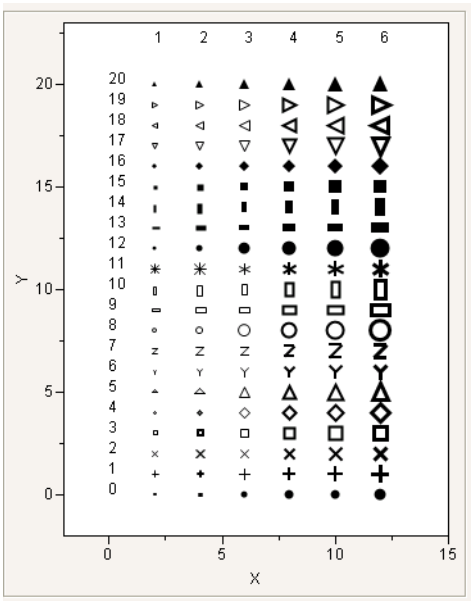
Markers

`Marker` draws a marker of the type that you specify (1–15) in the first argument at the point that you specify in the second argument. `Marker Size` scales markers from 0–6 (dot–XXXL). To set markers to the preferred size, use a value of -1.

```
ymax = 20;
New Window( "The markers",
   Graph Box(
      FrameSize( 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 12.12 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 these lines in the graph script above:

```
marker(i, color state(i), {j*2, i});
marker(color state(i), i, {j*2, i});
marker(combine states(colorstate(i),markerstate(i),hiddenstate(i)),{j*2, i});
```

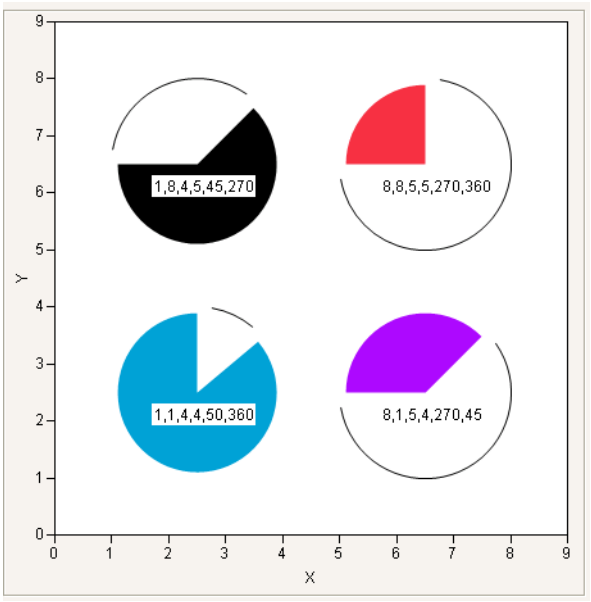
Again, points can also be specified as matrices of *x* and then *y* coordinates.

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.

```
New Window( "Pies and Arcs",
  Graph Box(
     framesize( 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 12.13 Drawing Pies and Arcs



Regular Shapes: Circles, Rectangles, and Ovals

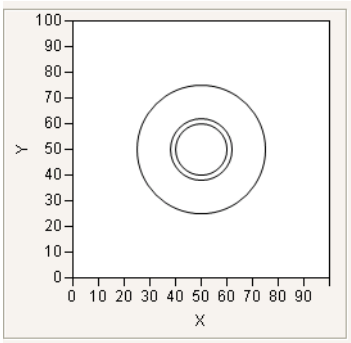
Circles

`Circle` draws a circle with the center point and radius given. Subsequent arguments specify additional radii.

```
New Window( "Circles",
   Graph Box( framesize( 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 12.14 Drawing Circles



Note that a circle is always a circle, even if you re-size 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:

```
New Window("Circles",
  Graph Box(framesize(200, 200),
     Circle({50, 50}, PixelRadius(10), PixelRadius(12), PixelRadius(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`*}).

```
New Window( "Rectangles",
  Graph Box( framesize( 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 a zero to get an unfilled rectangle, and a one to get a filled rectangle. The rectangle is filled with the current `fill color`. The default value for *`fill`* is 0.

Any negative fill argument produces an unfilled frame inset by one pixel:

```
New Window( "Framed rectangle",
  Graph Box( framesize( 200, 200 ),
     Rect( 0, 40, 60, 0, -1 )
```

```
    )
)
```

Ovals

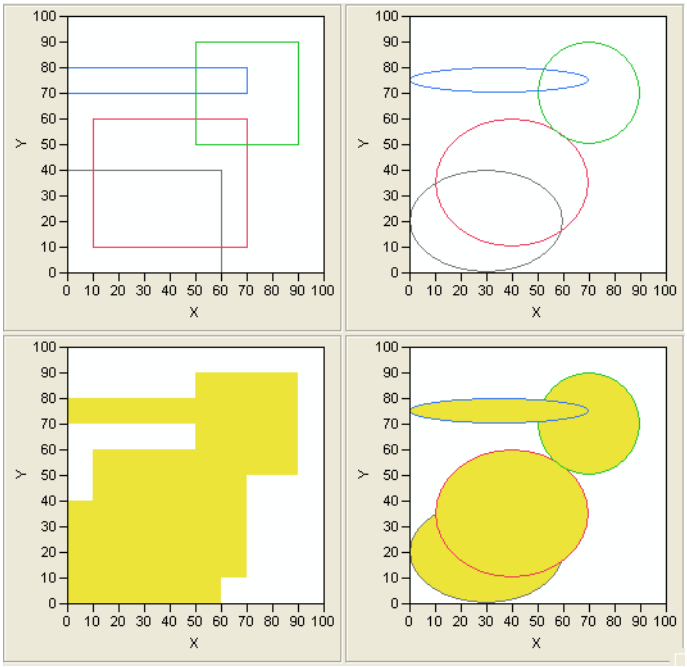
`Oval` draws an oval inside the rectangle given by its *x1, y1, x2,* and *y2* arguments:

```
new window("Ovals",
   graph box(framesize(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 uses the optional fifth argument, *`fill`*. Specify a zero to get an unfilled rectangle, and a one to get a filled oval. The oval is filled with the current `fill color`. The default value for *`fill`* is 0.

Figure 12.15 shows rectangles and ovals, drawn both filled and unfilled. Notice that filled rectangles do not have outlines, while ovals do. 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 12.15 Rectangles and Ovals, Unfilled and Filled



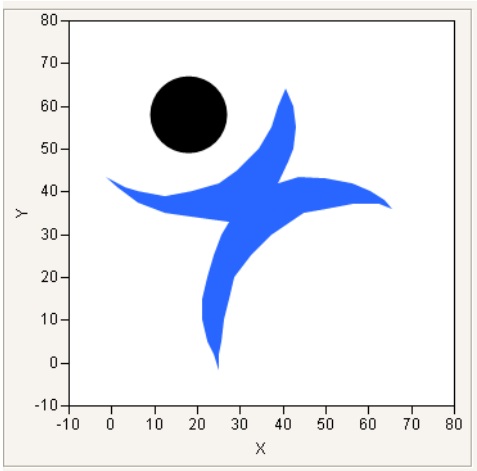
Irregular Shapes: Polygons and Contours

Polygons

`Polygon` works similarly to `Line`, connecting points but also returning 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 for `Marker`, above) 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, 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];
New Window( "The JMP man",
  Graph Box(
     framesize( 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 12.16 Drawing a Polygon



A related command, `In Polygon`, tells whether a given point falls inside the polygon specified. This code checks some points from the JMP man pictured in Figure 12.16.:

```
In Polygon(0,60, GcoordX,GCoordY);  //returns 0
In Polygon(30,38, GcoordX,GCoordY); //returns 1
```

Or you can add `In Polygon` to the JMP man script. Run this script, and then click various locations in the picture and watch the Log window.

```
new window("The JMP man",
   graph box(framesize(300,300), xscale(-10,80),yscale(-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. Its syntax is:

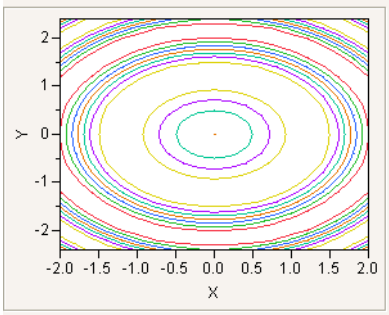
```
Contour(xVector,yVector,zGridMatrix,zContour,<zColors>);
```

Given an *n* by *m* matrix *zGridMatrix* of values on some surface, defined across the *n* values of *xVector* by the *m* values of *yVector*, this function draws the contour lines defined by the values in *zContour* in the colors defined by *zColors*.

```
// testContour

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 );
New Window( "Hat",
   Graph Box( X Scale( -2, 2 ), Y Scale( -2.4, 2.4 ), Contour( x, y, grid, z,
   zcolor ) )
);
```

Figure 12.17 Drawing Contour Lines



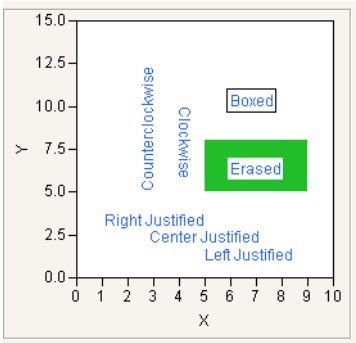
Adding text

You can use `Text` to draw text at a given location. 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 "erasing" whatever would otherwise obscure the text in a graph. It paints a background-colored rectangle behind the text. In the example below, notice how the erased text appears inside a white box over the green `Rect`. The other effects are self-explanatory.

```
mytext = New Window( "Text",
   Graph Box(
      framesize( 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 12.18 Drawing Text in a Graph Box



There is a variant of the text function that draws a string inside the rectangle specified by four coordinates specified as arguments, wrapping as needed. The syntax is

```
text( {left, top, right, bottom}, string)
```

Colors

Five commands control colors. `Fill Color` sets the color for solid areas, `Pen Color` for lines and points, `Back Color` for the background of text (similar to the box around the erased text above), `Background Color` for the graph's background color, and `Font Color` for added text.

Fill colors preempt pen colors for drawn shapes. You do not get both, as in some drawing packages. To get both a fill and a penline, draw two shapes, one with fill and one without. A color can be chosen with a single numeric argument, or a color name in quotation marks, or an RGB value. The standard colors can be chosen with numbers 0–15 (both 0 and 15 are black) or by their names.

Table 12.1 Standard JMP Colors

Number	Name	Demonstration script to try
0	Black	colors={"Black", "Gray", "White", "Red", "Green", "Blue", "Orange", "BlueGreen", "Purple", "Yellow", "Cyan", "Magenta", "YellowGreen", "BlueCyan", "Fuchsia"}; ymax=15; mygraph=New Window("The JMP colors", Graph Box(FrameSize(300,300), XScale(0, ymax), yScale(0, ymax+2), for(i=0, i<ymax, i++, If(colors[i + 1] == "White", Fill Color(65); Rect(0, i + 1 + .5, 15, i + 1 - .5, 1)); pen color(colors[i+1]); text color(colors[i+1]); hline(i+1); text({2, i + 1},i); text({5,i+1},colors[i+1]))));
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	

Larger numbers cycle through shading variations of the same color sequence. A script demonstrating this appears under "Colors and Markers" on page 354 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	dark shades
32–47	light shades
48–63	very dark shades
64–79	very light shades
80–84	shades of gray, light to dark

If you prefer to use RGB values, type a list with the percentages for each color in red, green, blue order.

```
pen color({.38,.84,.67}); // a lovely teal
```

`RGB Color` and `Color to RGB` convert color values between JMP color numbers and the Red-Green-Blue system. For example, to find 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.

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. For example,

```
Heat Color( z, <<{"", {{1, 1, 0}, {0, 0, 1}} })
Heat Color( z, <<{"", {blue, green, yellow} })
```

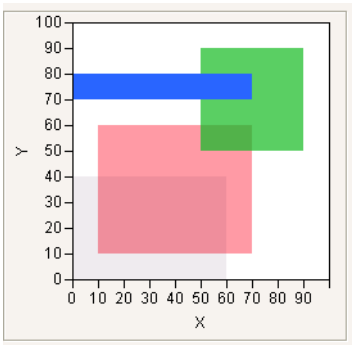
Transparency

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, while 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.

```
New Window( "Transparency",
  Graph Box(framesize( 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 12.19 Transparency and Rectangles



Fill patterns

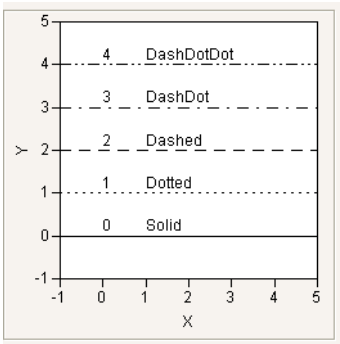
The `Fill Pattern` function has been deprecated and is now obsolete. It can be present in a script without causing errors, but has no effect.

Line types

You can also control `Line Style` by number (0–4) or name (`Solid`, `Dotted`, `Dashed`, `DashDot`, `DashDotDot`).

```
linestyles = {"Solid", "Dotted", "Dashed", "DashDot", "DashDotDot"};
New Window( "The line styles",
   Graph Box(
      FrameSize( 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 12.20 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.

```
pen size(2); //double-width lines
```

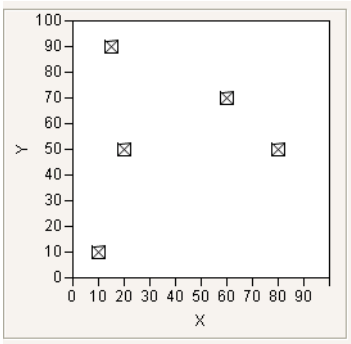
Drawing With Pixels

You can also draw using pixel coordinates. First you set the `Pixel Origin` in terms of graph coordinates, and then use `Pixel Move To` or `Pixel Line To` commands in pixel coordinates relative to that origin. The main use for `Pixel` commands 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 );
);
New Window( "Custom markers",
   Graph Box(
      framesize( 200, 200 ),
      ballotBox( 10, 10 );
      ballotBox( 15, 90 );
      ballotBox( 20, 50 );
      ballotBox( 80, 50 );
```

```
        ballotBox( 60, 70 );
    )
);
```

Figure 12.21 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

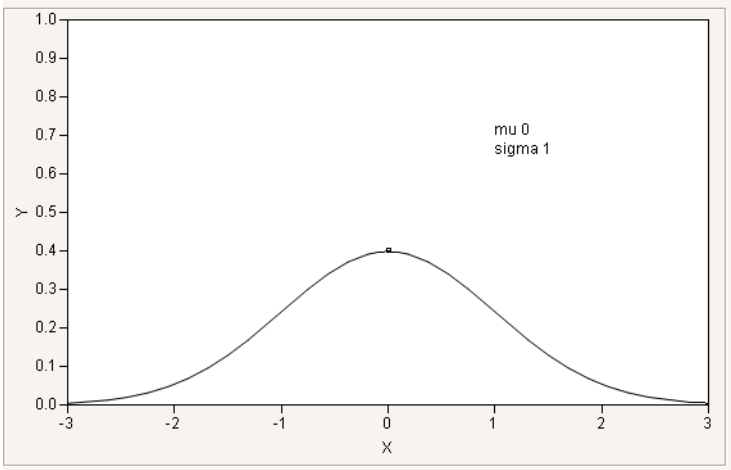
`Handle` places a marker at the coordinates given by the initial values of the first two arguments and draws the graph using the initial values of the arguments. You can then click and drag the marker to move the handle to a new location. The first script is executed at each mousedown to update the graph 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 "MouseTrap" on page 524.

```
// Normal Density
mu = 0;
sigma = 1;
rsqrt2pi = 1 / Sqrt( 2 * Pi() );
New Window( "Normal Density",
  Graph Box(
```

```
        FrameSize( 500, 300 ),
        X Scale( -3, 3 ),
        Y Scale( 0, 1 ),
        "Double Buffer",
        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 );
    )
);
```

In the sample Scripts folder, you can find scripts for showing the Beta Density, Gamma Density, Weibull Density, and LogNormal Density. The output for the normal is show below. Since we cannot show you the picture in motion, be sure to try this yourself.

Figure 12.22 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. For example:

```
YFunction(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.

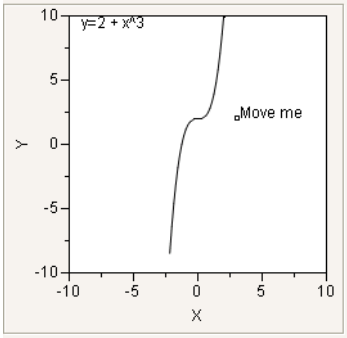
```
handle(a/2,b,a=2*x;b=y)
```

To generalize, suppose you define the `Handle` arguments as functions of the handle's (x, y) coordinates. For example, `a=f(x);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;
New Window( "Intercepts and powers",
   Graph Box(
      FrameSize( 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 12.23 Intercepts and Powers Example for `Handle`



`Handle` and `For` can be nested for complex graphs.

```
a=5; b=5;
New Window("powers",
   Graph Box(FrameSize(200,200),XScale(-10,10),yScale(-10,10),"Double Buffer",
   for(i=0,i<1.5,i+=.2,
      pen color(1+10*i);
      text color(1+10*i);
```

```
    YFunction(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))));
```

You can use more than one handle in a graph:

```
amplitude = 1; freq = 1; phase = 0;
NewWindow("Sine Wave",
  Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-5,5),"Double Buffer",
    YFunction(amplitude*sine(x/freq+phase),x);
    Handle(freq,amplitude,freq=x;amplitude=y);
    Handle(phase,.5,phase=x);
    Text({3, 4}, "amplitude: ", Round( amplitude, 4 ),
    {3, 3.5}, "frequency: ", Round( freq, 4 ),
    {3, 3}, "phase: ", Round( phase, 4 ))));
```

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`s and `Handle`s in a graph, put the `Handle`s before the `MouseTrap`s so they have 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;
New Window("Viewing a 3-D function in Flatland",
  Graph Box(FrameSize(300,300),XScale(-5,5),yScale(-5,5),DoubleBuffer,
  ContourFunction(exp(-(x-x0)^2)*exp(-(y-y0)^2)*(x-x0),x,y,z0/10);
  handle(-4.5,z0, z0=round(y*10)/10); // get the z-cut values from a handle
  vline(-4.5);text size(9);text(Counterclockwise,{-4.6,-4},
    "Drag to set the z-value for contour cut: z = " || char(z0/10));
  markersize(2);marker(2,{x0,y0});
  mousetrap(x0=x;y0=y); //set the origin to the click-point
  text({-4.25,-4.9},"Click any location to set the function's
  centerpoint.")));
```

You might use `MouseTrap` to collect points in a data table, such as for visually interpolating points in a graph. Here is an example illustrating a script that could be adapted and added to a data plot (such as a scatterplot from Fit Y by X) for that purpose:

```
dt = new Table("dat1");
Current Data Table(dt);
NewColumn("XX",Numeric);
```

```
NewColumn("YY",Numeric);
x=0; y=0;
add point = expr(
   dt<<addRows(1);
   row()=nrow();
   :xx = x;
   :yy = y);
NewWindow("Add Points",
   Graph Box(FrameSize(500,300),XScale(-5,5),yScale(-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. This means that if you click, drag, and release, the point that is added to your data set is the point where you let go of the mouse button, not the point where you pressed it down.

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, draws the ith list item at the ith (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:

```
dragMarker (xMatrix, yMatrix, dragScript, mouseupScript)
dragLine   (xMatrix, yMatrix, dragScript, mouseupScript)
dragRect   (xMatrix, yMatrix, dragScript, mouseupScript)
dragPolygon(xMatrix, yMatrix, dragScript, mouseupScript)
dragText   (xMatrix, yMatrix, "text", dragScript, mouseupScript)
```

They all must have L-value arguments for the coordinates, in other words, 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`.

The `Drag` operators 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 `Mouse Trap` 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.

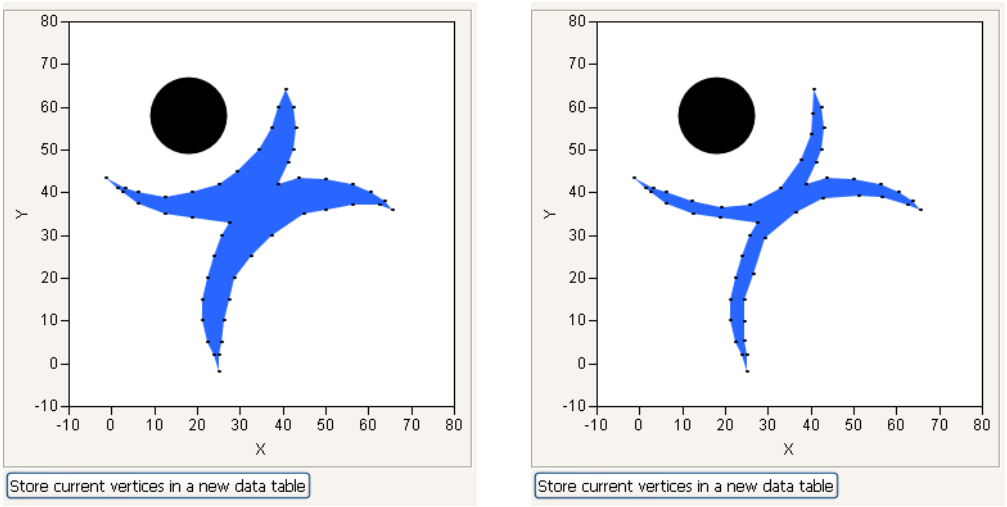
You could just as easily put *`storepoints`* in the fourth argument of `Drag Polygon` or `Drag Marker`, but that would create a data table after each drag, and you probably just want a single data table when you are finished. Regardless, the values in *`gCoordX`* and *`gCoordY`* update with each drag.

```
::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];
New Window( "Redraw the JMP Man!",
   V List Box(
      Graph Box(
         framesize( 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 )
```

```
    )
);
```

Perhaps you think the JMP Man needs to lose some weight. 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 12.24 Redraw the JMP Man



This example uses two operators discussed under "Constructing Display Trees" on page 423 in the "Display Trees" chapter:

- `Button Box`, which creates controls outside the graph,
- `V List Box`, which glued the graph box and the button box together in the same graph window.

Troubleshooting

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), and that the values make sense for the graph.

Creating 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. For more information about scripting, see the *Scripting Guide*.

To access the background map functionality through JSL, find an example in the Sample Data Library. For example, run JMP and select **Help > Sample Data Library** and open Pollutants Map.jmp. Edit the script called Bubble Plot Street Map. Look for the command in the script that begins with Background Map.

To figure out the correct parameters for the background map command, simply look at the background map window. The wording is the same as the JSL commands. To create a background map, simply use the command `Background Map()`.

There are two types of background maps that you can specify: `Images()` and `Boundaries()`. Each of these then takes a parameter, which is the name of the map to use. The name is one of the maps listed in the window.

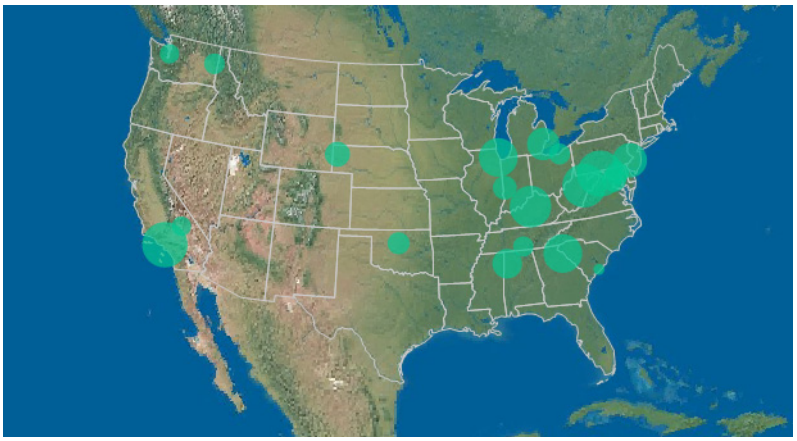
- For `Images()`, the choices are Simple Earth, Detailed Earth, NASA, Street Map Service, and Web Map Service. If you use Web Map Service, then there are two additional parameters: the WMS URL and the layer supported by the WMS server.
- For `Boundaries()`, the choices vary since boundaries can be user-defined. But a typical choice might be World.

To add a background map that uses the Simple Earth imagery and U.S. States as a boundary, the command would look like this (spacing is not important):

```
Background Map ( Images ( "Simple Earth" ), Boundaries ( "US States" ) )
```

Figure 12.25 shows the result.

Figure 12.25 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" ) )
```

You can also create your graph and add a background map through the user interface. Then from the red triangle menu use **Script > Save Script to Script Window** to see the script that is generated.

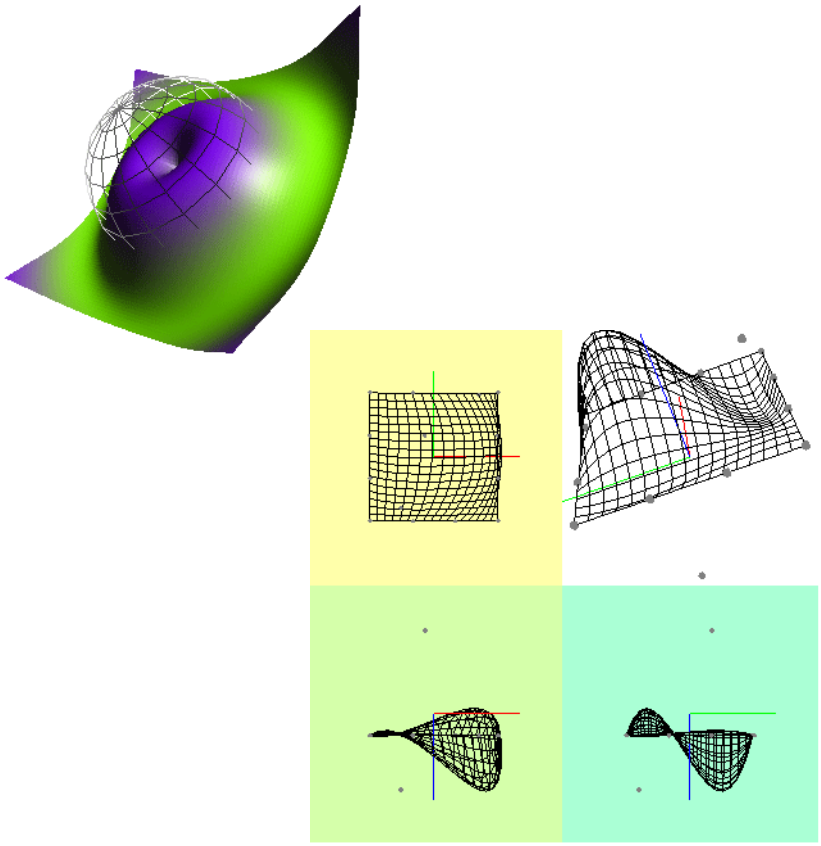
Chapter 13

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, and as such is not an implementation that is 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 ships 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 Web site opengl.org is a good jumping off point 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 your scenes can be journaled and played back later, and provides a pick 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.

OpenGL is a trademark of Silicon Graphics, Inc.

JSL 3-D Scene Boxes

These commands are necessary to set up and configure a 3-D scene.

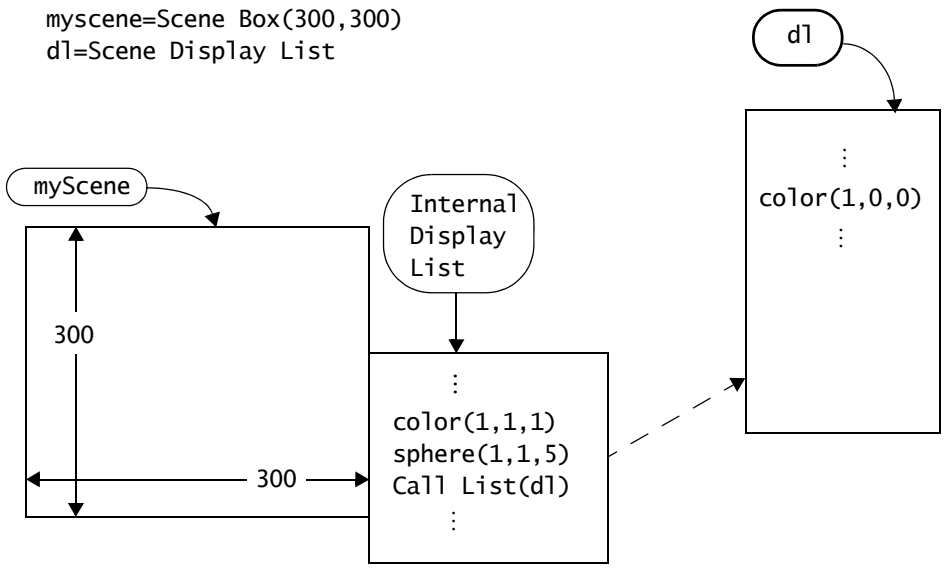
Like all displays in JMP (detailed in the "Display Trees" chapter on page 401), 3-D scenes must be placed in a display box (in this case, a `Scene Box`). This box is then placed in a window. Therefore, a simple 3-D scene script has the following form.

```
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



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 = SceneBox( 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. Since the "Hello World" text is to be drawn at the origin (0, 0, 0), the `Translate` command is added to the display list to move 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.

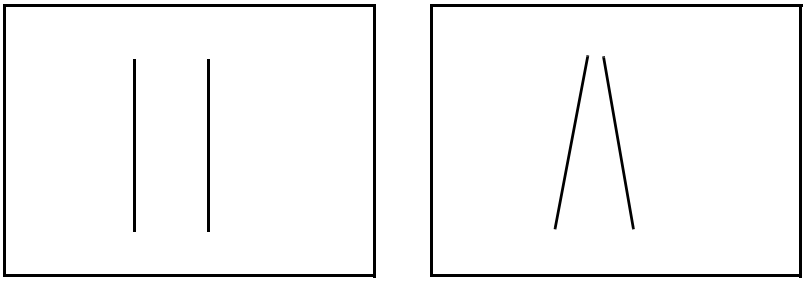
```
// create a display list and send it commands
greeting = Scene Display List();
greeting << color(1,0,0); //set the RGB color of the text
greeting << Text( center, baseline, 0.2, "Hello, World." ); //add text

//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.

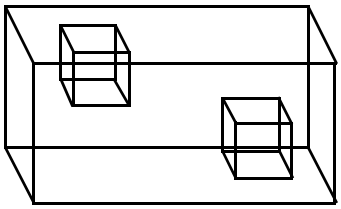
Figure 13.4 Parallel Lines in an Orthographic Projection(left) and a Perspective Projection (right)

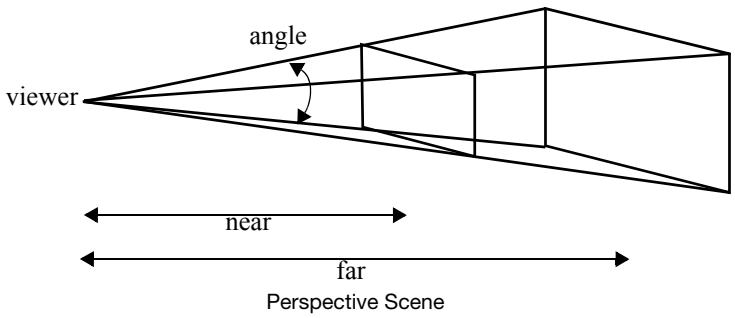


As another example, imagine looking at a tube edge-on (like a telescope). In an orthographic projection, the tube would appear as a thin circle. In a perspective projection, the circle would have a thickness; the hole at the far end of the tube would appear smaller than the close hole, and the interior of the tube is visible.

Therefore, the viewable space of an orthographic projection is a rectangular shape, while 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





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 so that the rendering engine can 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, while 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

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 )`.

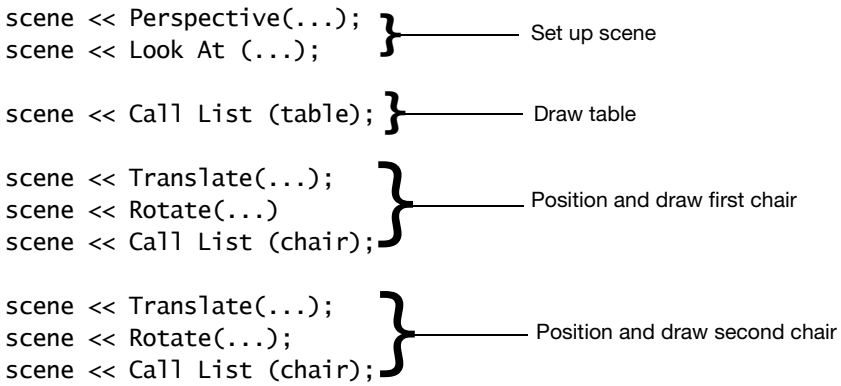
You can also specify the three axis values in a matrix. For example, `Rotate( 90, [1, 0, 0] )`.

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:

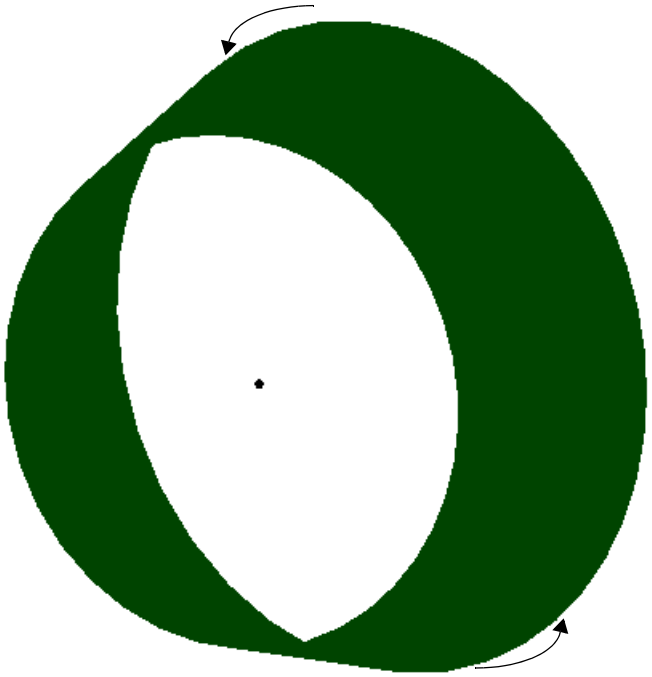
Figure 13.7 Using Translate and Rotate



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.

```
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 600, 600 );

// put the scene in a window.
NewWindow( "Example 1", scene );

for (i=1, i<360, i++,
   scene << clear;

// the lens is 45 degrees, near is 1 units from the camera, far is 10.
   scene << Perspective( 45, 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 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. Because the model is typically constructed at the origin, 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 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 on the sphere's surface and drag it around, thus causing the scene to rotate.

Use an `ArcBall` instead of a `CallList` 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 556 for details of pushing and popping.)

For example, examine the script from "Primitives Example" on page 546. 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 popup 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.

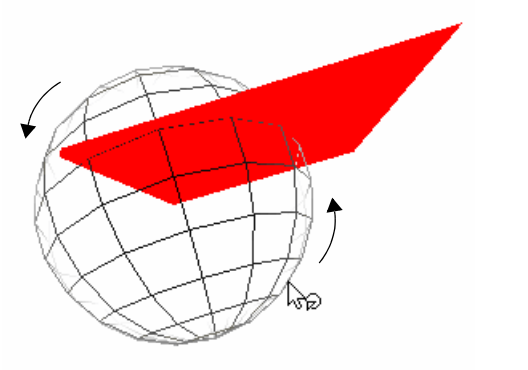
Note: The ArcBall does not have to be showing to react to mouse commands. It is shown here for display purposes only.

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

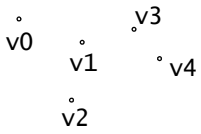
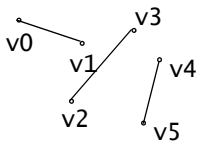
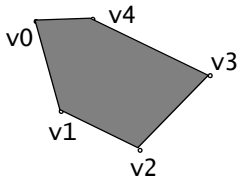
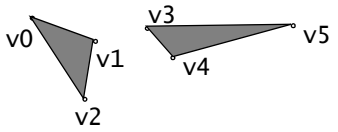
primitive type=POINTS	Draws a point at each of the vertices.	
primitive type=LINES	Draws a series of (unconnected) line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If n is odd, the last vertex is ignored.	
primitive type=POLYGON	Draws a polygon using the points v0,..., 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.	
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.	

Table 13.1 Primitive Types *(Continued)*

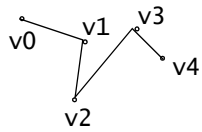
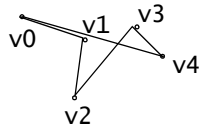
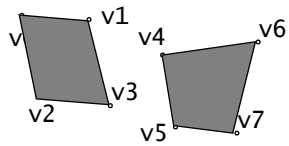
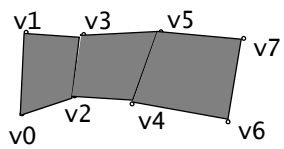
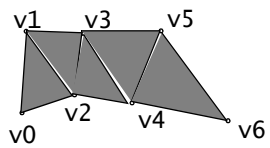
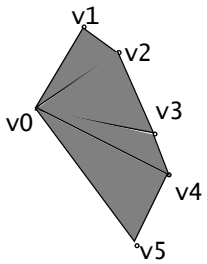
primitive type=LINE_STRIP	Draws a line segment from v0 to v1, then from v1 to v2, and so on. Therefore, n vertices specify n–1 line segments. Nothing is drawn unless there is more than one vertex. There are no restrictions on the vertices describing a line strip; the lines can intersect arbitrarily.	
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.	
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.	

Table 13.1 Primitive Types *(Continued)*

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.

```
// create a display list and send it commands
shape = Scene Display List();
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` function 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

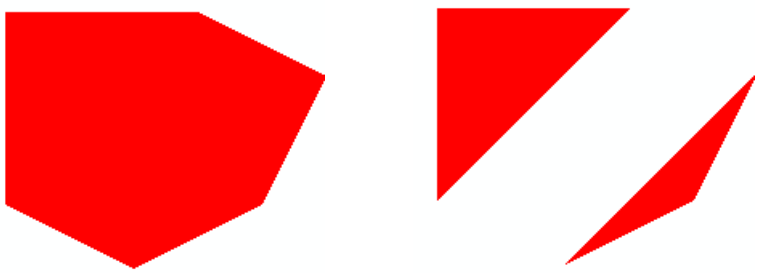
```
shape <<Begin(POLYGON);
```

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)`.

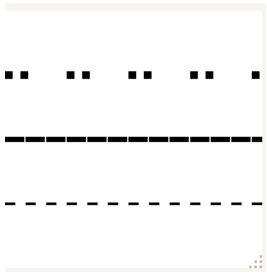
The *factor* argument expands each binary digit to two digits. In the example above, `Line Stipple(2, 255)` would result in 00000000000000001111111111111111.

For example, the following script draws three lines, each of different widths (the `Line Width` commands) and stippling patterns.

```
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 200, 200 );

// put the scene in a window.
New Window( "Stipples", scene );

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, and lines in the model change length on the screen even though nothing changes in model units.

Fill Pattern

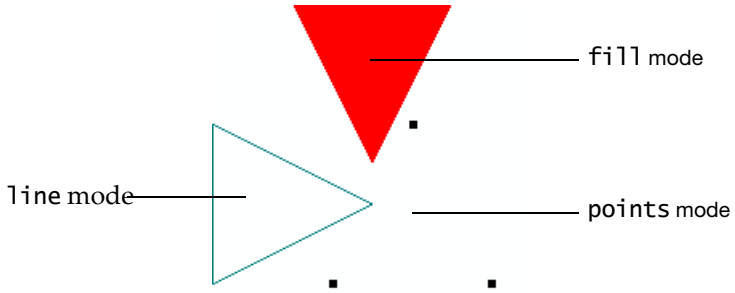
Polygons are rendered with both a front and a back, and the drawing mode of each side is customizable. This enables the user to see the difference between the back and front of the polygon.

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

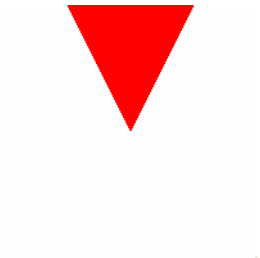
Code from above script	comments
`shape = Scene Display List();`	Creates a display list
`shape << Begin(TRIANGLES);` `shape << Vertex(0, 0, 0);` `shape << Vertex(-1, 2, 0);` `shape << Vertex(1, 2, 0);` `shape << End();`	Creates a display list named `shape` that holds vertices for the triangles. All the 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);` `scene << Update;` `// update the scene to see the triangle`	Draw the first triangle in red. 

Table 13.2 Translate and Rotate Commands *(Continued)*

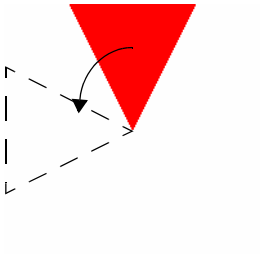
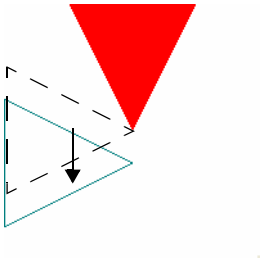
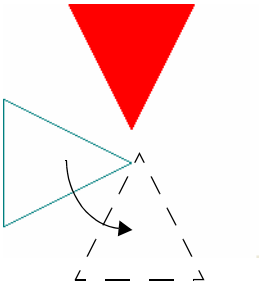
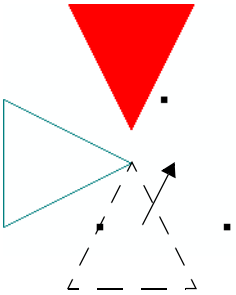
Code from above script	comments
`scene << Rotate (90, 0, 0, 1);` `scene << Translate (-0.5, 0, 0);` `scene << Color(0, 0.5, 0.5);` `scene << Polygon Mode(FRONT_AND_BACK, LINE);` `scene << Call List(shape);` `scene << Update;` `// update the scene to see the triangle`	Draw the second triangle in teal. Note that we first rotate the triangle.  And then translate it. 

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);` `scene << Point Size(5); //large points so they are visible` `scene << Polygon Mode(FRONT_AND_BACK, POINT);` `scene << Call List(shape);` `scene << Update;` `// update the scene to see the triangle`	Draw the third triangle as black points. First rotate.  And then translate to get the final picture. 

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`*), where *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 tailor the automatic lighting.

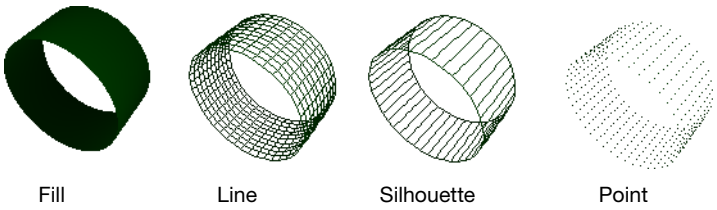
`Quadric Normals(`*`mode`*`)` tells what type of normal should be automatically generated. The argument *`mode`* can be `None`, `Flat`, or `Smooth`. `Flat` makes faceted surfaces. `Smooth` makes the normals at each vertex be the average of the adjacent polygons.

`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



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.

```
/* make a scene box...holds an OpenGL scene */
scene = SceneBox( 600, 600 );

/* put the scene in a window */
NewWindow( "Example 2", scene );

scene << Perspective( 45, 3, 7 );
  /* the "lens" is 45 degrees, near is 3 units from the camera, far is 7 */
scene << Translate( 0.0, 0.0, -4.5 );
  /* move the world so 0,0,0 is visible in the camera */
scene << Rotate( 30, 0, 1, 0 );
  /* rotate the first text about the Y (vertical on screen) axis */
scene << Color( 1, 0, 0 );
  /* pure red */
scene << Text( "center", "baseline", .2, "First Red String" );
scene << Translate( 0.0, 0.0, -2.0 );
  /* the next string is even farther away from the camera */
scene << Rotate( 30, 0, 1, 0 );
  /* rotate the second text about the Y (vertical on screen) axis */
```

```
scene << Color( 0, 1, 0 );
  /* pure green */
scene << Text( "center", "baseline", .2, "Second Green very long string" );
scene << Update;
  /* update the displaybox in the window using the current display list */
```

Figure 13.14 Rotating and Translating Text Strings

Note the green string is extending backwards beyond the far clipping plane. Change the 7 to 10 in the `Perspective` command to see the complete string.

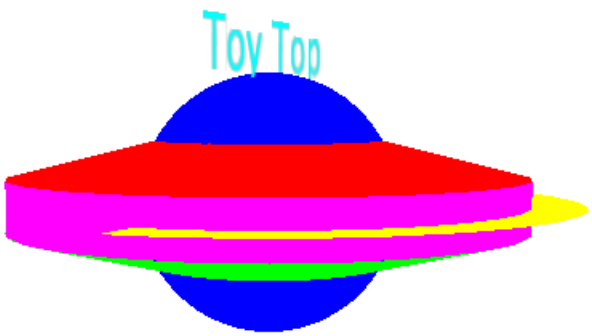
Using the Matrix Stack

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 = SceneDisplayList();
toyTop<<PushMatrix;
   toyTop<<Translate(0,0,.1);
   toyTop<<Color(1,0,0); // red
   toyTop<<Cylinder(1,.2,.2,25,5);
      /* baseRadius, topRadius, height, slices, stacks */
toyTop<<PopMatrix;
toyTop<<PushMatrix;
   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<<PopMatrix;

toyTop<<Color(0,0,1); // blue
toyTop<<Sphere(.5,30,30);
   /* radius, slices, stacks */

toyTop<<Color(1,1,0); // yellow
toyTop<<PartialDisk(1,1.2,25,2,0,270);
   /* innerRadius, outerRadius, slices, rings, startAngle, sweepAngle */

toyTop<<PushMatrix;
   toyTop<<Translate(0,0,-.1);
   toyTop<<Color(1,0,1); // magenta
   toyTop<<Cylinder(1,1,.2,25,3);
   /* baseRadius, topRadius, height, slices, stacks */
toyTop<<PopMatrix;

toyTop<<PushMatrix;
   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<<PopMatrix;

/* make a scene box...holds an OpenGL scene */
scene = SceneBox( 600, 600 );

/* put the scene in a window */
NewWindow( "Example 3", scene );
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 << CallList( toyTop );
```

```
/* update the displaybox */
scene << Update;
```

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);
gl<<Mult Matrix(xt*xr*yr*zr); // order of multiplication matters with matrices
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 only exists 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

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 (lightn)` 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 only apply 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.

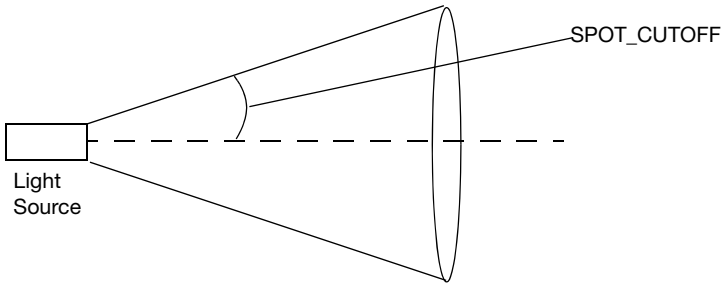
Light in the real-world decreases in intensity as distance from the light increases. Since a directional light is infinitely far away, it does not make sense to attenuate its intensity as a function of distance. However, JSL attenuates a light source by multiplying the contribution of the source by an 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 560 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 and 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)
```

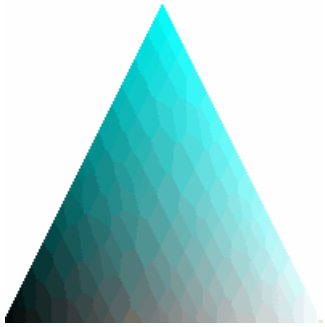
where *mode* can be `SMOOTH` (the default) or `FLAT`. `SMOOTH` shading interpolates the colors of the primitive from one vertex to the next. `FLAT` mode duplicates the color of one vertex across the entire primitive.

The following script changes the color at each of a triangle's vertices. The `FILL` shade model interpolates the color of the interior automatically.

```
// make a scene box...holds an OpenGL scene.
scene = SceneBox( 200, 200 );

// put the scene in a window.
```

```
NewWindow( "Shade Model", scene );
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

Table 13.5 Material Arguments and Default Values *(Continued)*

Argument	Default Value	Meaning
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.

Complete details of 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)
```

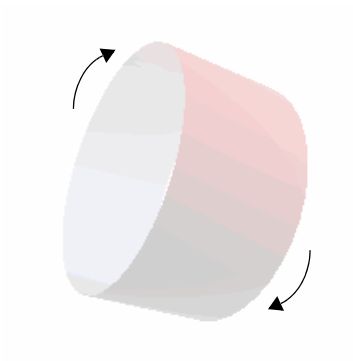
Example

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 = SceneBox( 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 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); )
```

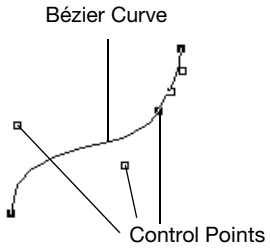
Figure 13.18 Fog



Bézier Curves

A complete discussion of Bézier curves is beyond the scope of this book. 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 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

NPOINTS = 4;
  /* 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 */

points = J(NPOINTS, 3, 0);
  // create an array of x,y,z triples
for( x = 1, x <= NPOINTS, x++,
   points[x, 1] = (x-1)/(NPOINTS-1) - .5;
     // x from -.5 to +.5
   points[x, 2] = randomuniform() - .5;
     // y is random in same range
   points[x, 3] = 0;
     /* z is always zero, which causes the curve to stay in a plane */
);

spline = SceneBox(boxwide,boxhigh);

spline << ortho( -.6, .6, -.6, .6, -2, 2 );
  /* data from -.5 to .5 in x and y; this is a little larger */
```

```
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, points );
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(points[i,1], points[i,2], points[i,3]);
  spline<<End;
  spline<<Push Matrix;
  spline<<Translate(points[i,1], points[i,2], points[i,3]);
  spline<<Text(center, bottom, .05,char(i));
  spline<<Pop Matrix;
);

New Window("Spline", spline);
```

http://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 566 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 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; /* only cares 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 control. `K`=1 for the Shift key, `K`=2 for the Control (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.

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	

Chapter 14

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 Datafeed for capturing real-time data from a laboratory instrument
- using SAS, MATLAB, or R through JMP Scripting Language (JSL)
- using JMP with Excel
- connecting to databases
- controlling JMP externally by OLE automation
- parsing XML

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 269 in the "Programming Methods" chapter.

Contents

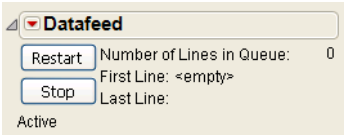
Real-Time Data Capture

A Datafeed is a real-time method to read data continuously, such as from a laboratory measurement device connected to a serial port. A Datafeed object sets up a concurrent thread with a queue for input lines that arrive in real time and are processed with background events. You set up scripts to interpret the 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 Datafeed Object

To create a DataFeed object, use the `Open DataFeed` command with arguments specifying details about the connection:

```
feed = Open DataFeed( options );
```

No arguments are required. You can simply create the Datafeed 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` command 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 Datafeed 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 = Data Feed[2];
```

Options

(Windows only) 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 Datafeed 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: For Datafeed purposes, a line of data is a single value (a datum). A line is not to be confused with a data table row, which can contain values for many variables on a subject.

Set Script attaches a script to the Datafeed object. This script is triggered by the On DataFeed handler 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 << getLine ) ) );
```

A Datafeed 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 (!IsMissing(x), current data table()<<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 Datafeed with Messages

A Datafeed 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 Datafeed 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 Data Feed`. However, it would be more common to send them as messages to a Datafeed object that is already present.

You can send lines to a Datafeed from a script. This is a quick way to test a Datafeed. Include a text argument or a global that stores text:

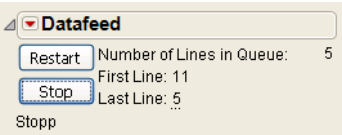
```
feed << Queue Line( "14" );
feed << Queue Line( myValue );
```

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" );
```

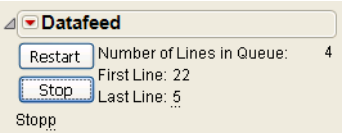
Figure 14.2 Datafeed: 5 Lines Queued



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"
```

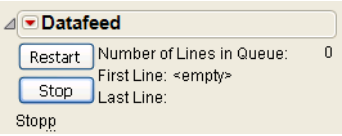
Figure 14.3 Datafeed: 4 Lines Queued



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 Datafeed and its window:

```
feed << Close;
```

To disconnect from the live data source:

```
feed << Disconnect
```

Examples of Datafeed

Reading Data

Here is a typical Datafeed 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 ),
         Current Data Table( ) << 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.

```
// make a data table with one column
dt = New Table( "Gap Width" );
dc = dt << New Column( "gap", Numeric, Best );

// set up control chart properties
dt << Set Property( "Control Limits", {XBar( Avg( 20 ), LCL( 19.8 ),
   UCL( 20.2 ) )} );
dt << 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 << AddRow( {:gap = z} )
   );
);
feed << SetScript( feedScript );
```

```
// start the control chart
Control Chart(
   Sample Size( 5 ),
   K Sigma( 3 ),
   Chart Col(
      gap,
      XBar(
         Connect Points( 1 ),
         Show Points( 1 ),
         Show Center Line( 1 ),
         Show Control Limits( 1 )
      ),
      R(
         Connect Points( 1 ),
         Show Points( 1 ),
         Show Center Line( 1 ),
         Show Control Limits( 1 )
      )
   )
);

// 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 ) )
);
```

Store the Script in a Data Table

You can further automate the production setting by placing a Datafeed 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 Datafeed script and saves data to a new data table file.

Table 14.1 DataFeed Messages

Message	Syntax	Explanation
`Open Data Feed` `Data Feed`	`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. `Data Feed` is a synonym.

Table 14.1 DataFeed Messages *(Continued)*

Message	Syntax	Explanation
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 Datafeed queue.
Get Lines	feed << Get Lines	Returns as a list and removes all lines from the Datafeed queue.
Queue Line	feed << Queue Line(*string*)	Sends one line to the end of the Datafeed queue.
Stop	feed << Stop	Stops processing queued lines.
Restart	feed << Restart	Restarts processing queued lines.
Close	feed << Close	Closes the Datafeed 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 Datafeed queue but leaves the Datafeed object active.

Dynamic Link Libraries (DLLs)

Note: 64-bit JMP cannot load 32-bit DLLs, and 32-bit JMP cannot load 64-bit DLLs. If you have a 32-bit DLL that you have used successfully in the past, but you are now using 64-bit JMP, you must recompile your DLL for 64-bit 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 define it and then 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(...)
```

The named arguments for `Convention` are as follows:

- `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 DLL section of the *JSL Syntax Reference* for the `Declare Function` message arguments.

Finally, use the `UnLoadDLL` message to unload the DLL:

```
dll_obj << UnLoadDLL
```

Example

Suppose that you want the script to load a Windows 32-bit DLL or a 64-bit DLL based on the user's computer. This example tests for the Windows operating system and then the computer processor's bit level.

```
If( Host is( Windows ),
  If(
```

```
      Host is( Bits64 ),
      // Load the 64-bit DLL.
         dll_obj = Load DLL( "c:\Windows\System32\user32.dll" ),
      // Load the 32-bit DLL.
         dll_obj = Load DLL( "c:\Windows\SysWOW64\user32.dll"),
      // If neither DLL is found, stop execution.
         Throw
      );
   dll_obj <<DeclareFunction(
      "MessageBoxW",
      Convention( STDCALL ),
      Alias( "MsgBox" ),
      Arg( IntPtr, "hWnd" ),
      Arg( UnicodeString, "message" ),
      Arg( UnicodeString, "caption" ),
      Arg( UInt32, "uType" ),
      Returns( Int32 )
   );
   result = dll_obj << MsgBox(
      0,
      "Here is a message from JMP.",
      "Call DLL",
      321
   );
   Show( result );
);
dll_obj << UnLoadDLL
```

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;
```

For Simple DLL Functions

If your DLL contains functions that are very simple, the `Call DLL` message declares the specified function with the specified signature, as shown in Table 14.3.

```
dll_obj <<Call DLL(function_name, signature, arguments)
```

Table 14.3 Signature Values

Signature	Argument is of Type
"v"	void
"c"	string
"n"	int

Then, the function is called, passing in the *argument*.

Example

```
// Load the DLL, given the specified path
dll_obj = Load DLL( "D:\Release\MyDLL.DLL" );
// Call the function inside the DLL named MyExportedFcn.
// This function takes one numeric as input and in this case,
// the value of the input argument is to be 654.
dll_obj << CallDLL( "MyExportedFcn", "n", 654 );
dll_obj << UnLoadDLL();
```

Effectively, what these two lines accomplish is to call `MyExportedFcn`, passing 654 as input to the function. Conceptually, it is as if the JSL script executed `MyExportedFcn(654);`

Using Sockets in JSL

Another tool that can be useful in establishing a live Datafeed 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`).

The JMP Samples/Scripts folder 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", "www.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

JMP supports ODBC access to SQL databases through JSL with the Open Database command.

```
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: Database table names that contain the characters $# -+/%()&|;? are not supported.

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 ODBC connection dialog box
- "DSN=" and then the data source name and any other information needed to connect to the data source.

For example:

```
"DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/<version number>/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"
   ```

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");
```

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));` or go to **File > Preferences > Tables** and select the **ODBC Hide Connection String** option.See *Using JMP* for details.

Creating a Database Connection and Executing SQL

You can use the following functions to handle more complex database operations:

```
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.

Example

Open a connection to your database:

```
dbc = Create Database Connection(
  "DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/<version number>/Samples/
  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);
```

Working with SAS

JMP has several ways of interacting with the SAS system.

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,

```
Current Data Table( ) << 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”
```

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.

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 more information, see the *Using JMP*.

Make the Connection

First, use the `Metaconnect` command 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 `SAS Get Lib Ref` command to view the libraries on that server:

```
librefs = SAS 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:

```
librefs = SAS Assign Lib Refs("MyLib", "c:\public\data");
```

Open SAS Data Sets

First, assign a SAS library reference:

```
SAS Assign Lib Refs("MyLib", "c:\public");
```

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=SAS 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:

```
dt=SAS Import Data(“MyLib”, “PURCHASES”);
```

or

```
dt=SAS Import Data(librefs[1], datasets[12]);
```

or

```
dt=SAS 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:

```
bookvars=SAS 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.

```
dt=SAS Import Data(librefs[1], datasets[12], columns(bookvars[1],
   bookvars[2], bookvars[4]));
```

Save SAS Data Sets

To save a JMP data table or an imported SAS Data Set, use the `SAS Export Data( )` command:

```
SAS 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:

```
SAS Submit("proc print data=sashelp.class; run;");
```

Two optional arguments control whether you see the output and the SAS log in JMP:

```
SAS Submit("SAS Code" <,No Output Window(True|False)> <,Get SAS
   Log(True|False)>);
```

You can also see the SAS Log at any time using the command

```
SAS Get Log( );
```

`SAS Get Log( )` returns the contents, which can be placed in a JSL variable and used like any JSL string.

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 Sample/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 on 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.

Working with MATLAB

MATLAB, a product of MathWorks Inc., provides an interactive working environment for analyzing and visualizing computational models. MATLAB is available for Windows (both 32-bit and 64-bit), Macintosh OS X, and Linux (64-bit). JMP supports MATLAB on both Windows and Macintosh 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 MATLAB Integration Functions section in the *JSL Syntax Reference* book for details on using MATLAB functions in JMP.

Textual output and error messages from MATLAB appear in the Log window.

Installing MATLAB

MATLAB must be installed on the same computer as JMP.

Because JMP is both a 32-bit and a 64-bit Windows application, you must install the corresponding 32-bit or 64-bit version of MATLAB. For the supported version of MATLAB, see the JMP website: http://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.

Working 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:

http://cran.r-project.org

Because JMP is supported as both a 32-bit and a 64-bit Windows application, you must install the corresponding 32-bit or 64-bit version of R. For the supported version of R, see the system requirements on the JMP website: http://www.jmp.com/support/system_requirements_jmp.shtml

Override Default R Install Location

Normally JMP determines R_HOME internally if it is not defined in the Windows system registry:

64-bit R Install Computer\HKEY_LOCAL_MACHINE\SOFTWARE\R-code\R\InstallPath

32-bit R Install

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\R-code\R\InstallPath

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");
```

How JMP Finds R

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`

   For 32-bit JMP running on a 64-bit machine, the `InstallPath` value is under the following key:

   `HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432NODE\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.

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 14.4 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 14.4 Equivalencies Between JMP and R Data Types for R Send( )

JMP Data Type	R Data Type
Numeric	Double
String	String
Matrix	Double Matrix
List	List
Data Table	Data Frame
Row State	Integer
Datetime	Date and Time
Duration	Time/Duration

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 14.5 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 14.5 Equivalencies Between 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

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;

//Initiate the R connection
R Init();

//Send the Here variable to R
//Here:x creates the R object Here.x"
R Send( Here:x );
R Submit( "print(Here.x)" );
/* Note that the JMP log labels the output with the original JMP variable
   reference Here:x. */

//::y creates the R object y
R Send( ::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" ) );
R Submit( "print(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.*/

//z creates the R object z
```

```
R Send( 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 Submit( "class(X)" );
/* R output is:
[1] "list"
*/

R Submit( "Y<-unlist(X)
     class(Y)" );
/* Object Y is now a character vector. The R output is:
[1] "character"
*/

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); // Sends 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.

Working with Excel

You can script the Profiler interface to Excel, although not the Transfer to JMP interface. The basic syntax is as follows:

```
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 Excel Profiler object, you can send the following messages to run the JMP prediction profiler using the Excel model. For example:

```
excel_obj <<Prediction Profiler( 1 );
```

Parsing 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.

The following example illustrates

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, // assume first row is 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 ) ), // assume 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 in JMP/12/Documentation for details about automating JMP. This document introduces how to automate JMP through Visual Basic and using Visual C++ with MFC. It also contains details for the methods and properties that JMP exposes to automation clients like Visual Basic and Visual C++

The JMP Samples/Automation folder contains several example Visual Basic .Net, Visual C# .Net, and Visual C++ .Net programs that automate features in JMP.

Automating JMP through Visual Basic

Starting a JMP Application

The first step in automating JMP is to start it up. However, it's important to look at the resources available to help you with the JMP methods and properties. JMP provides a type library that allows automation controllers like Visual Basic (VB) to display a list of the methods and properties that JMP exposes, along with arguments that the methods require. This library is called JMP.TLB.

There are two steps to make the JMP type library available to VB.

1. Select **Project > References** in VB. A list of applications that are known to VB appears. If JMP is not in that list, select **Browse**. A file window asks you to locate a .tlb (Type library) file. Find the icon for the JMP type library in the JMP directory. Select this library and click **OK**.
2. Open the object browser by selecting **View > Object Browser** in VB. Select JMP from the drop down list box.

Now you can see the JMP automation classes and constants. You can now select a class, and the methods available to that class appear in the right list box for the object browser. If you select a method, a short helper string appears at the bottom of the window. This string lists the arguments for the method. Constants are used when methods require a restricted set of arguments, typically denoting a specific action.

Now that you have access to the type library information, write the necessary code to instantiate JMP. This is done with `CreateObject`. In global declarations for the VB project, create a variable of type *`JMP.Application`*. This is done as:

```
Dim MyJMP As JMP.Application
```

Now dimension some other variables. Good examples are `DataTable`, `Distrib`, `Oneway`, and `JMPDoc`. These are specified with `JMP.DataTable`, `JMP.Distribution`, `JMP.Oneway`, and `JMP.Document` respectively.

To create a JMP session, make it visible, and load a data table, add the following code to your VB script.

```
Dim JMPDoc As JMP.Document
Set MyJMP = CreateObject("JMP.Application")
MyJMP.Visible = True
Set JMPDoc = MyJMP.OpenDocument("C:\Program Files\SAS\JMP\<version
   number>\Samples\Data\Big Class.jmp")
```

The `Dim` statement indicates the type of variable. This declaration should go in the general declarations section of your VB project, though. If you do not do this, the JMP objects are destroyed when the variable goes out of scope at the end of the procedure.

JMP comes up invisible by default, as required by automation guidelines. Therefore, one of your first moves should be to make it visible, as shown in the above code.

Launching an Analysis

Now that you have a data table open, you can launch an analysis and manipulate it. Each analysis must first be created. Then, the required arguments for the analysis must be specified. Optional settings can also be specified. Then the analysis is launched. Additional option processing can then be done on the analysis object after the launch.

```
Dim Oneway As JMP.Oneway
Set Oneway = JMPDoc.CreateOneway
Oneway.LaunchAddY ("Height")
Oneway.LaunchAddX ("Age")
'Set an option before the launch
Oneway.Quantiles (True)
'Create the initial analysis output
Oneway.Launch
Oneway.MeansAnovaT (True)
Oneway.MeansStdDev (True)
```

```
Oneway.UnequalVariances (True)
Oneway.NormalQuantilePlot (True)
Oneway.SetAlpha (0.05)
Oneway.Save(JMP.OnewaySaveConstants.oscCentered)
Oneway.Save(JMP.OnewaySaveConstants.oscStandardized)
Oneway.CompareMeans(JMP.OnewayCompareConstants.occAllPairs, True)
Oneway.CompareMeans(JMP.OnewayCompareConstants.occEachPair, True)
```

The first step is to create the analysis object, which is done by calling the `CreateOneway` method of the document class. Next, X and Y columns are selected, and then `Launch` is called to create the actual One-way analysis. Each analysis platform has a distinct creation method, which you can view under the `Document` object in the object browser. In many cases, it is possible to specify options before the `Launch` of the object, so the analysis output uses the options that are already set. In this example, most option processing is done after the launch of the analysis, which shows the options popup in the display. As you can see, most methods are a simple setting of options, like you might do from a menu. `SetAlpha` takes an argument, since you do not want to open a window for interaction during automation. `CompareMeans` takes two arguments, one for the type of comparison and one for the toggle to indicate on or off. The `Save` method takes a predefined constant (viewable in the object browser) that tells the Oneway analysis what to save.

Most analysis methods work this way, although some like Bivariate produce additional objects when methods are called. An example is:

```
Set Fit = Bivar.FitLine
Fit.ConfidenceFit (True)
Fit.ConfidenceIndividual (True)
```

Here, the `FitLine` method produces an object of type Fit. This object has methods and properties of its own, which can be manipulated. Remember, the new object created by `FitLine` can be manipulated only while its variable is in scope.

If a method produces an object that can also be automated, the object browser indicates this. For `FitLine`, the object browser specifies that the return type is `As Fit`.

Since this is not a predefined type like short or BSTR, you can probably guess that this is an object. If you look farther down the object browser, you see `Fit` as an object type. This confirms that an object is produced, and also gives you the methods that `Fit` supports.

Creating and Populating a Data Table

New data tables can be created with the (appropriately named) `NewDataTable` method of the `Application` object. A filename is assigned at creation time. This method returns a column object, which must be retained as long as you want to add rows. By default, 20 rows are created. The `SetCellVal` method can be used to populate individual cells, and `AddRows` can be used to add rows as needed. Here is an example:

```
Dim Col As Object
```

```
Set DT = JMP.NewDataTable("C:\test.jmp")
Set Col = DT.NewColumn("Col1", dtTypeNumeric, 0, 8)
DT.Visible = True

'You must add rows before populating the table with data
DT.AddRows 20,0

'Set Cell values to increments of 1.5
For i = 1 To 10
  Col.SetCellVal i, i * 1.5
Next i
DT.Visible = False
For i = 11 To 20
  Col.SetCellVal i, i * 1.5
Next i
DT.Visible = True

'This adds 5 rows to the end of the table
DT.AddRows 5, 0
'This adds 5 rows after row 2
DT.AddRows 5, 2

'Now save the data table using the previously specified filename
DT.Document.Save

'If you wanted to create a subset of the table, with only rows 1-3
'you could do the following
'Note: you could also create subsets using specific columns by adding the
'columns to a list using the AddToSubList member function of Datatable
Dim NewDT As JMP.Datatable
Dim DTDoc As JMP.Document
DT.SelectRows 1,3
Set NewDT = DT.Subset

'Now save the new table
Set DTDoc = NewDT.Document
DTDoc.SaveAs("C:\MySubset.jmp")
```

Example Programs

The JMP Samples\Automation folder contains several example Visual Basic .Net, Visual C# .Net, and Visual C++ .Net programs that automate features in JMP. The Visual Basic programs require Visual Studio 2005 or later.

The ANALYSIS example program shows simple automation cases for almost all of the JMP platforms. The example code tests the features of a platform, but it does not pretend to do

meaningful statistical analyses. Its purpose is for teaching automation coding. It is recommended that you make the JMP type library visible to the VB project. The first section of this document describes this process, which lets you see the methods and properties exposed by the automation platforms within JMP.

Likewise, the DATATAB example shows how to exercise the methods available for data table automation. No attempt is made to produce meaningful output.

The TIMPORT program shows the steps necessary to get a text file imported into JMP as a data table. Once this has been done, the data table can be manipulated just like the example in DATATAB, and analyses can be performed on the data just like in the ANALYSIS program.

The ODBCDemo program shows a simple example of importing a dBase file into JMP using ODBC access.

The WordDemo program shows the commands necessary to take a graphic section from a JMP report, copy it to the clipboard, and then insert it into a Microsoft Word document.

The FitModel and DOE examples show operators that are specific to those areas of JMP, and whose platform operator differs slightly from other platforms.

The sample code for all five example programs assumes the data files reside in the JMP Samples/Data directory. If you move your sample data files, you need to change the path information in the VB samples.

If there are differences between this document's examples of Visual Basic code and that in the sample programs, preference should be given to the sample program code.

An Example: Automating JMP From Excel 2007

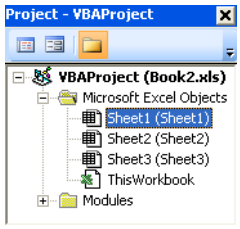
This example automates JMP using a macro within an Excel 2007 worksheet. The macro code is written in Visual Basic. It starts JMP in a visible state when the Excel worksheet is initially opened. The Excel worksheet is then imported into JMP using the ODBC automation interface. Once the worksheet data is in JMP, changes to individual worksheet cells are sent to JMP and changed in the JMP data table.

The first time a row value in Excel changes, JMP generates a Control Chart. Subsequent changes to the excel worksheet result in changes to the Control Charts. This is because Control Chart output is dynamically linked to the JMP data table, which in this example is dynamically updated by Excel. Every fifth time the Excel worksheet changes, a method is called in JMP to generate a .PNG file for the Control Chart. This allows users without JMP to view the output through a web browser. Finally, when the Excel worksheet closes, JMP shuts down through automation.

Begin by opening Microsoft Excel. To create a Visual Basic script for an Excel workbook, select **Visual Basic** from the **Developer** ribbon. The Visual Basic editor opens in a separate window. On the left side of the Visual basic editor, there is a pane entitled **VBA Project**. This pane shows

the sheets that might have Visual Basic code associated with them, as well as the workbook itself.

Figure 14.5 VBA Project for Excel



Code written for the workbook usually works for any of the sheets within the workbook.

There are three sections involved in the coding for this example. First, there are some variables that are global in scope that are declared in the `module1.bas` file. This allows these variables to be referenced in other code modules. A module can be inserted into the Visual Basic project by context-clicking on the VBA project icon and selecting **Insert > Module**. Type the following code into the module. The code declares instances of a JMP application, a JMP data table, and a flag to keep track of whether a document is open or not.

```
Public MyJMP as JMP.Application 'The JMP Application Object
Public DT As JMP.DataTable 'The JMP Data Table object
Public DocOpen as Boolean 'A flag indicating "JMP Table Open"
```

The next segment updates JMP when cells in the Excel worksheet change. It is called automatically because Excel generates the `Worksheet_change` event whenever a cell is changed, deleted, or added.

The Excel VBA Project Browser shows the sheets that are currently part of the workbook. The code below should be placed in the sheet that sends data to JMP. Double-click on the sheet icon in the VBA Project Window to bring up the code for that particular sheet.

```
Private Sub Worksheet_change(ByVal Target as Range)
  Dim Col as JMP.Column

  If(DocOpen) Then
    If(Target.Row = 1) Then
      Return
    End If
    If(DT.NumberRows < Target.Row - 1) Then
      DT.AddRows Target.Row - DT.NumberRows - 1, Target.Row
    End If
    If(Not IsArray(Target.Value) And Not IsEmpty(Target.Value)) Then
      Set Col = DT.GetColumnByIndex(Target.Column)
      Col.SetCellVal Target.Row - 1, Target.Value
    End If
```

```
    End If
End Sub
```

This code first checks to make sure JMP has a data table open. If the change is happening to the first row, then it is ignored because this is the column name in JMP. So, if a column name is changed in Excel, the corresponding change is *not* reflected in JMP. Code that would deal with heading changes could be inserted here, but is omitted in this example.

Next, if the row that has changed is beyond the number of rows that JMP is currently tracking in the data table, then the `AddRows` method is called to create more rows.

Finally, if the operation is on a single value and does not seem to signal a deletion, the JMP data table cell value is changed to the value that is passed into `Worksheet_Change`.

The main module is associated with the workbook. In the VBA Project Browser, the workbook code area is typically assigned the name `ThisWorkbook`, but this name can be easily changed. The following code goes into this area.

```
'Public(Global Variables) that all Workbook subroutines can access
Public Counter As Integer 'counter to update Control Chart every 5 changes
Public JMPDoc As JMP.Document 'instance of JMP Document
Public CChart As JMP.ControlChart 'instance of Control Chart
Public ChartOpen as Boolean 'Flag to set if chart is open
Public DB As AUTODB

'Shut Down JMP before closing the workbook
Private Sub Workbook_BeforeClose(Cancel as Boolean)
  DocOpen = False
  MyJMP.Quit
End Sub

'As soon as the workbook is opened via File Open, load JMP for Automation
Private Sub Workbook_Open( )
  Set MyJMP = CreateObject("JMP.Application") 'Create an instance of JMP
  MyJMP.Visible=True 'Make this instance of JMP visible
  Counter = 0 'initialize counter that counts changes
  DocOpen = False 'no document open yet
  ChartOpen = False 'no charts open yet, either

  'CHANGE THIS PATH TO POINT TO THE EXCEL WORKSHEET
  Set DB = MyJMP.NewDatabaseObject
  DB.Connect ("DSN=Excel Files;DBQ=C:\Book2.xls;")
  Set DT = DB.ExecuteSQLSelect("SELECT * FROM ""Sheet1$""")
  DB.Disconnect
  Set JMPDoc = DT.Document
  DocOpen = True 'Set flag to say that the document is open
End Sub
```

```
'This is the most important part.
'After the first piece of data has been changed, generate a control chart.
'After every 5 changes to Excel worksheet cells, generate a new PNG of the
  Control Chart.

Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Source As Range)
  Counter = Counter + 1
  'Save the control chart to a PNG every time 5 elements get updated
  If (Counter Mod 5 = 0 Or Counter = 1) Then

     'If the Control Chart has not been created yet, do so
     If Not (ChartOpen) Then
        Set CChart = JMPDoc.CreateControlChart 'create chart
        CChart.LaunchAddProcess "Column 1" 'Add column
        CChart.LaunchAddSampleUnitSize 5
        CChart.LaunchSetChartType jmpControlChartVar
        CChart.Launch 'launch the chart
        ChartOpen = True 'set flag to remember that a chart is open
     End If
     CChart.SaveGraphicOutputAs "C:\ControlChart.png", jmpPNG
  End If
End Sub
```

The `Workbook_Open` subroutine is called when the Excel table is initially loaded. It initializes some variables, starts JMP, and tells JMP to open (through ODBC) the same Excel file that is currently loaded into Excel 2007. Note that JMP opens the Excel file as a database object rather than opening it as a file. This is necessary because JMP does not open a file that is already open in another application.

The `Workbook_Change` event is generated every time a user changes the data in any cell in any worksheet in the workbook. This sample assumes that there is only one active worksheet in the workbook. The first time the user changes a cell value in the worksheet, the `Workbook_Change` subroutine creates a Control Chart in JMP using the current data table.

In this sample, the `Workbook_change` subroutine also creates a PNG graphic file of the Control Chart output and updates it on the disk every fifth time a change is made to the workbook. This just gives some ideas on how Excel events and JMP automation can be used together to create output.

Finally, the `Workbook_BeforeClose` subroutine is invoked when the Excel workbook is closed, but before the window goes away. The code within this subroutine instructs JMP to close down as well.

Note that there are some limitations in this method. This example is good if the only activities that occur with the data are additions or changes. The Excel `Worksheet_Change` event is very limited in the reporting that it provides. In particular, cell-by-cell updating of a JMP data table can be difficult in instances where deletion, drag and drop, or block replication needs support.

If these are problem cases, it is probably better to rely on a brute-force approach. One way is to reload the data into JMP every time a certain number of changes occur. An example is shown here.

```
Private Sub Workbook_SheetChange(ByVal Sh as Object, ByVal Source as Range)
  Counter = Counter + 1
  If (Counter Mod 10 = 0) Then
     'If there is a previous chart of Table opened, close it first
     If(DocOpen) Then
        JMPDoc.Close False, ""
        CChart.CloseWindow
     End If

     Set JMPDoc = MyJMP.OpenDocument(InstallDir + "C:\BOOK1.XLS")
     Set DT = JMPDoc.GetDataTable
     DocOpen = True

     'Now, create the control chart.
     'This one is keyed to the data in "Column 1".
     'If 5 or more values are changed,
     'JMP should generate a new Chart and save it as a
     'PNG file to disk.
     'The PNG file can be viewed with Internet Explorer.

     Set CChart = JMPDoc.CreateControlChart
     CChart.LaunchAddProcess "Column 1"
     CChart.LaunchAddSampleUnitSize 5
     CChart.LaunchSetChartType jmpControlChartVar
     CChart.Launch
     CChart.SaveGraphicOutputAs "C:\ControlChart.png", jmpPNG
  End If
End Sub
```

This sample reloads the data every time there are 10 changes to the Excel Workbook. First, it removes JMP Control Charts and data tables that were previously created. Next, it loads the new data and creates a Control Chart.

This sample works best for small amounts of data. If very large Excel files are involved, this approach is not efficient because of the reloading of the table into JMP.

Automating JMP through Visual C++

Using C or C++ to create an automation client can be a long, tedious task. However, if you use the support provided by MFC in Microsoft Visual C++, the task is considerably easier. There are several steps that must be performed in order to get to a state where you can launch the automation server application (JMP in this case). The `AutoClient` application that is included

in the JMP Samples/Automation/Visual C++ Sample directory contains some code that provides ideas on how to get started. The Microsoft sample application CALCDRIV also shows a MFC-based automation client. CALCDRIV is typically included with Visual C++, and on MSDN CDs.

`AutoClient` shows how to start up JMP and drive a Bivariate analysis and the data table. The sample is much smaller than any of the Visual Basic samples. However, the mechanics behind all the automation calls that you might want to use are the same as the examples with Bivariate and the data table. The following steps are based on the Visual C++ Version 5.0 UI.

Steps for Automating JMP

1. Create your application, either manually or through App Wizard. Specify support for OLE automation. Even if you are not automating your own application, you need to include the OLE headers and initialization code. If you are retrofitting an existing application, you need to make sure that you include OLE support. This usually means including afxole.h in your application, and calling `AfxOleInit()` in your application InitInstance routine. Consult the MFC OLE documentation for details about this.
2. Bring up the Class Wizard and select the Automation tab. Select the **Add Class** drop down list and then the **From a Type Library** option. Navigate to the JMP install directory until you find **JMP.TLB**. Select this type library.
3. You are prompted to confirm the classes that you want to use in your project. If you are unsure what objects (and interfaces) that you want, select them all by Shift-clicking. Select the names for the files where the class wizard generates interface stubs and header information. Class Wizard is generating wrapper classes based on the MFC ColeDispatchDriver class. This gives you easy access to the OLE **Invoke** automation function without having to know a lot of the technical details. Select **OK**. Class Wizard generates the two files (.h and .cpp). You should include the .h file in whatever .cpp files use the JMP automation objects. For example, your View class implementation file.
4. The Class View of your Workspace now shows the Interface classes that you have imported. You can examine the methods and properties for each class through this class view.
5. To start JMP, define a variable of type *`IJMPAutoApp`* that persist for the length of the automation session. Call `CreateDispatch` on this variable, passing in the JMP ProgID (“JMP.Application”) as the lone argument. At this point, when the code executes JMP starts.
6. Call `SetVisible(TRUE)` on the JMP object created in step 5. If you do not want to see JMP execute, do not do this step. However, for debugging it is necessary.
7. Now you can use the JMP application object to spawn further objects, which themselves can spawn more objects. The first thing you probably want to do is load a Data table. To load an existing JMP data table, call the `OpenDocument` method on the JMP object created in

step 5. If successful, this method returns a dispatch pointer that can be attached to an object of type IJMPDoc using the `AttachDispatch` method.

8. The `IJMPDoc` object provides the methods to launch the analysis and graphing platforms. Once you create an analysis and attach the dispatch pointer, you can specify the data table columns to use in the analysis and then you can launch it. Once the analysis is launched, you can manipulate it using the properties and methods specific to that particular type of analysis. Code that is taken from the sample application that describes steps 5–8 is shown below:

Example Program

```
//Note, no error handling is done in this example
IJMPAutoApp m_DispDriver;
IJMPDoc m_Doc;
IAutoBivar m_Bivar;
IAutoFit m_FitLine;

//Create the initial dispatch driver that uses the IJMPAutoApp
//interface specification (taken from jmpauto.h)
m_DispDriver.CreateDispatch("JMP.Application");

if (m_DispDriver)
{
  //If JMP successfully started, make it visible
  m_DispDriver.SetVisible(TRUE);

  //Now open a data table as a document. The document interface
  //pointer that is returned is then attached to our Doc dispatch
  //driver class that uses the IJMPDoc interface specification.
  m_Doc.AttachDispatch(m_DispDriver.OpenDocument(
     "C:\\JMPDATA\\BIGCLASS.JMP"));
}
  //First, call CreateBivariate on the Doc interface to create
  //a dispatch object to a Bivariate analysis. If there is already
  //a previous dispatch interface in m_Bivar, MFC releases it
  //in AttachDispatch.
  m_Bivar.AttachDispatch(m_Doc.CreateBivariate( ));

  //Now add Height and Weight as the columns to analyze
  m_Bivar.LaunchAddX("Height");
  m_Bivar.LaunchAddY("Weight");

  //Launch the analysis
  m_Bivar.Launch( );
```

```
//Create a FitLine. Since the Fit can be automated, attach the dispatch
//pointer that is returned from FitLine( ) to a DispatchDriver object
m_FitLine.AttachDispatch(m_Bivar.FitLine( ));

//Now do a few more fits. This example does not automate these fit
//objects, although they do support automation.
m_Bivar.FitPolynomial(3.0);
m_Bivar.FitSpline(1000.0);

//Now manipulate the first FitLine object
m_FitLine.ConfidenceFit(TRUE);
m_FitLine.ConfidenceIndividual(TRUE);
```

Chapter 15

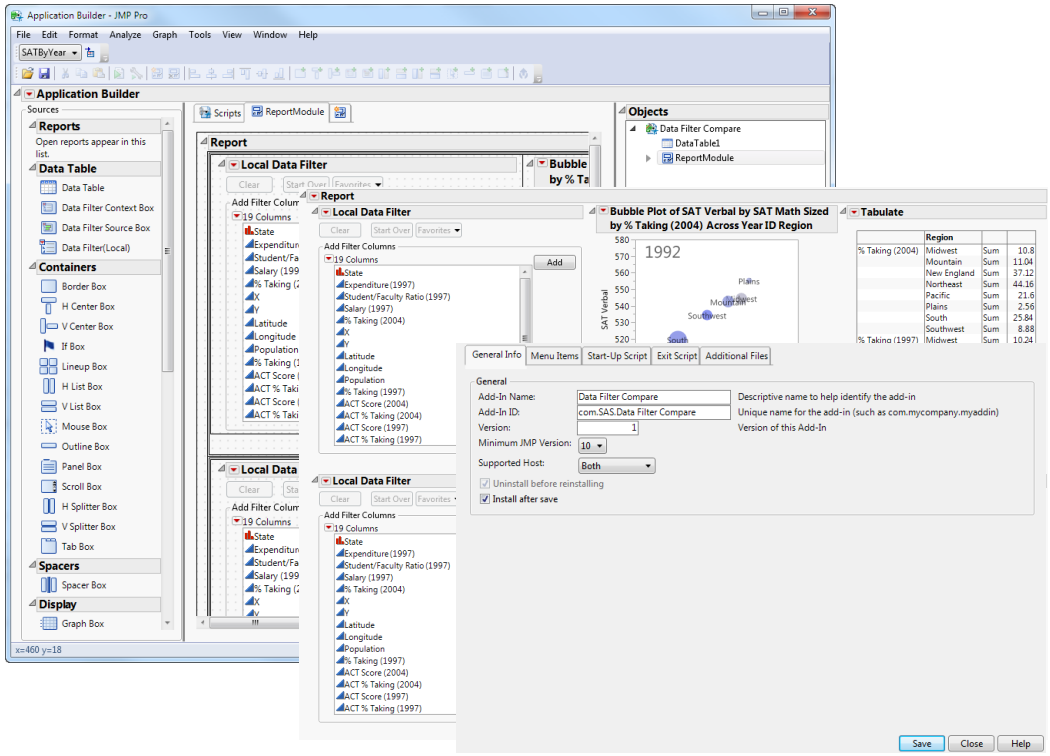
Creating and Sharing Applications

Application Builder and Add-In 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 15.1 The Path from Application Builder to Customized Reports to a JMP Add-In



Contents

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. Then you write scripts to control the functionality of each object.

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.

Example

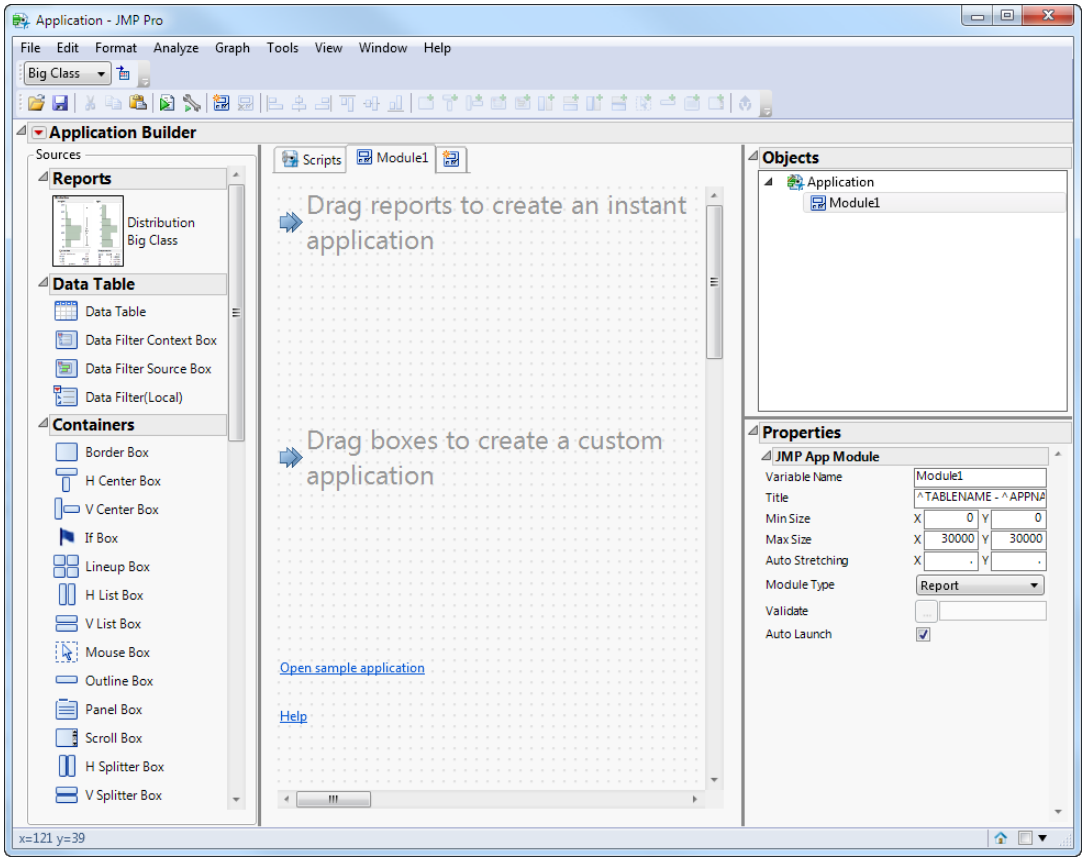
Application Builder helps you quickly create instant applications without writing JSL. One example is a JMP window in which you arrange reports, graphs, and other objects to make them easy to interact with at once.

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 Macintosh).

   The Application Builder window appears.

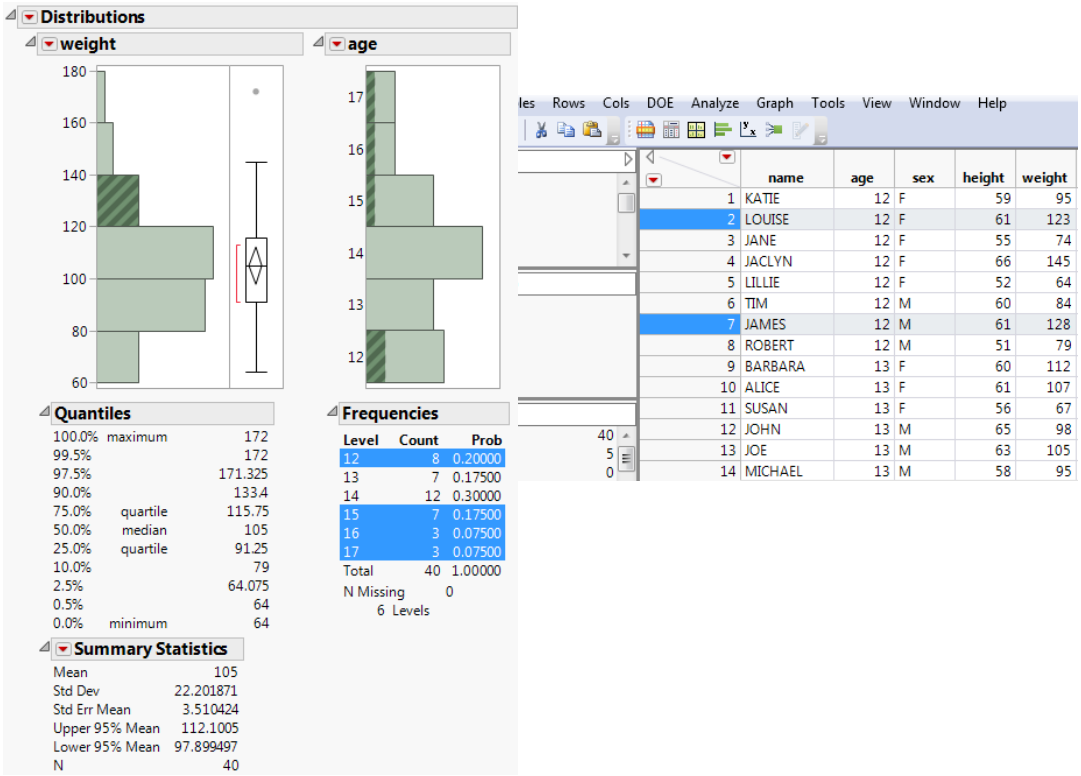
Figure 15.2 New Application with Distribution Report Displayed in the Sources Pane



4. In the Sources pane, drag the Distribution report onto the application workspace.
5. Right-click the report on the workspace and select **Move to Corner.**
6. From the Application red triangle menu, select **Run Application**.

   The Distribution report appears in a new window (Figure 15.3). 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 15.3 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.

Table 15.1 describes the terms that 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.

Table 15.1 Application Builder Terms

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.

Table 15.1 Application Builder Terms *(Continued)*

Instant Application	An application that typically consists of reports and does not require JSL scripting.
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 237 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 pane 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.

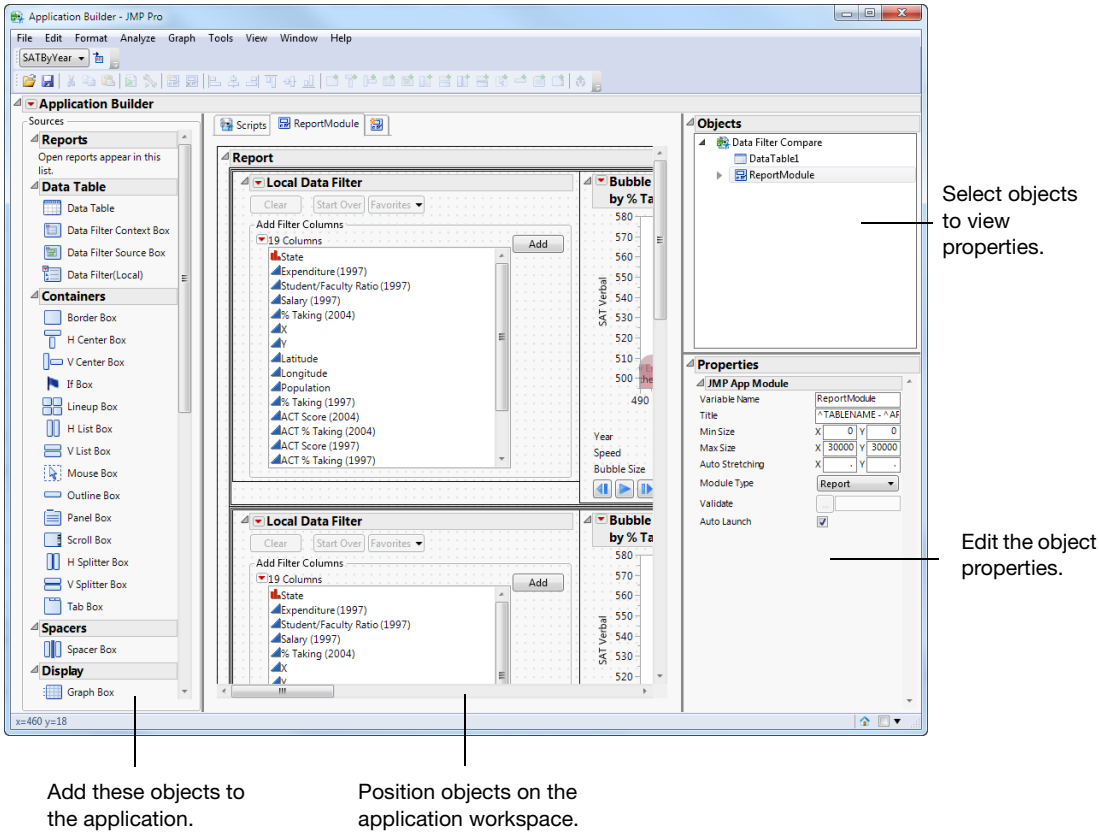
Scripting Depending on the complexity of the application, you write JSL scripts. See "Write Scripts" on page 639 for details about writing scripts for an application.

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 15.4 shows an application under development in Application Builder.

Figure 15.4 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 pane 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 pane 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 pane, 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.

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 68 in the "Scripting Tools" chapter for more information.

Open Sample Opens a sample application from the JMP Samples/Apps folder. These samples show you how to set up common applications, and you can modify them as you wish. Table 15.2 describes the samples.

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 Sources Shows or hides the Sources panel.

Show Objects and 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.

Script Lets you save the application to a data table, journal, script window, or add-in. See "Options for Saving Applications" on page 642 for details.

Table 15.2 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.

Table 15.2 Sample Applications Installed with JMP *(Continued)*

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 Measurement Systems Analysis (MSA) Combo Chart.jmpappsource	Creates a collection of reports for Measurement System Analysis (MSA).
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

Create a Blank Application

1. Select **File > New > Application** (or **File > New > New Application** on Macintosh).

   The Application Builder window appears.
2. Select **File > Save** and save the file as a JMP Source File. The extension is .jmpappsource.

Combine Open Windows to Create an Application

You can create an application from open data tables or reports rather than beginning with a blank application.

1. Select the windows that you want to combine.
   - On Windows, select the check box in the lower right corner of the open data table or report window. Select **Combine selected windows** next to the check box.
   - On Macintosh, select **Window > Combine Windows**, select the open data table or report window, and then click **OK**.

   The objects appear in a new report window.
2. From the report's red triangle menu, select **Edit Application**.

For more information about combining windows in JMP, see *Using JMP*.

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 pane, 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 pane. See "Customize Object Properties" on page 636 for details.
4. Add scripts to scriptable objects. See "Write Scripts" on page 639 for details.
5. (Optional) To prevent a module from launching when you run the application, select the module in the Objects pane and deselect **Auto Launch**. (You might do this while testing one out of multiple modules.)
6. From the Application Builder red triangle menu, select **Run Application** to test your application.

   Your application appears.

Create a Module

1. Select **Format > Add Module.**
2. In the Properties pane, select the **Module Type**:
   - Dialog
   - Dialog with Menu
   - Modal Dialog
   - Launcher
   - Report

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 pane 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 pane. This method is particularly helpful when the internal object completely covers the container, as with a border that has an internal text box.

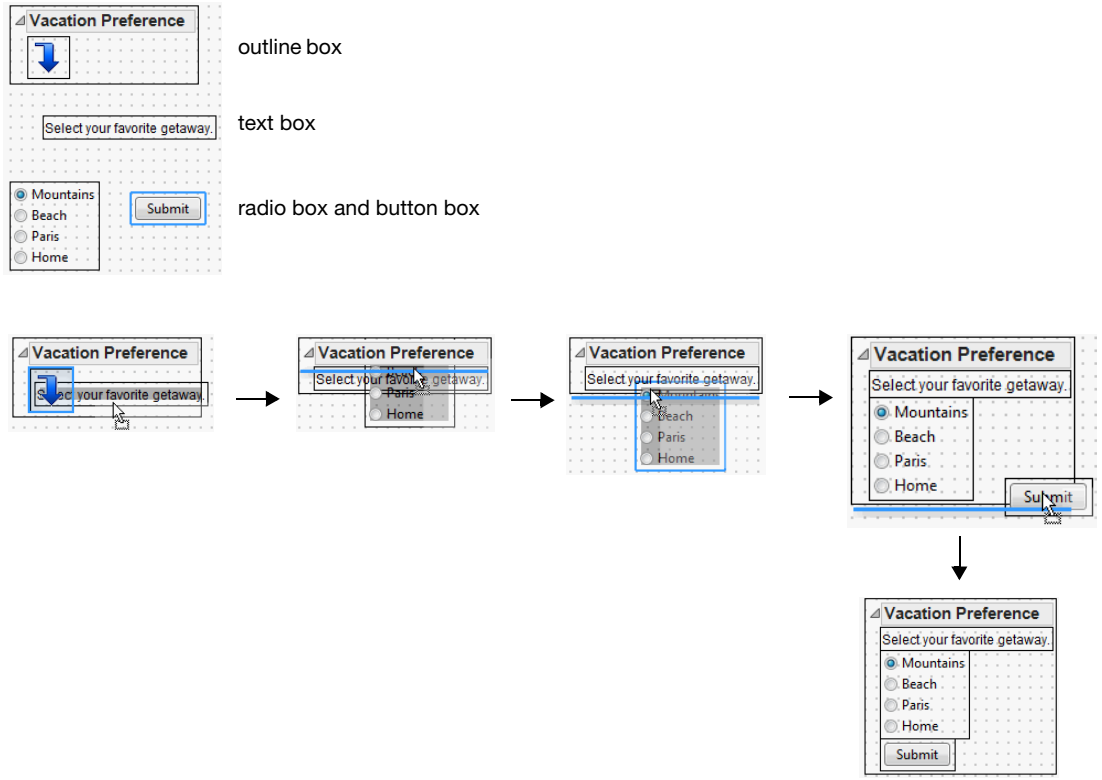
Drag an Object

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, deselect **Snap to Grid** in the Application Builder red triangle menu. You can also turn off **Snap to Grid** for all applications in the JMP Preferences. (Select **File > Preferences > Platforms > JMP App** to change the option.)
- 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 15.5 shows the various methods for dragging objects inside a container.

Figure 15.5 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 the TAB key 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 Macintosh, hold down CTRL and 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.

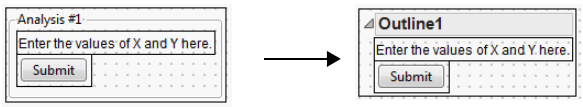
Tips:

- 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** (Figure 15.6).

Figure 15.6 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 15.7).

To insert selected objects into a container, click the appropriate button on the toolbar or select **Format > Add Container**.

Figure 15.7 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 Macintosh) and dragging the object also creates duplicates of the originals.

Delete an Object

- Remove one or more objects by selecting them and then pressing the DELETE key.
- Drag the objects outside of the module.
- On Macintosh, select the objects, hold down CTRL and COMMAND, and then select **Delete**.
- On Windows, select **Edit > Clear** to delete all objects in a module.
- Select the object and then press the BACKSPACE key.

If you no longer need the object's script, you can delete the script also.

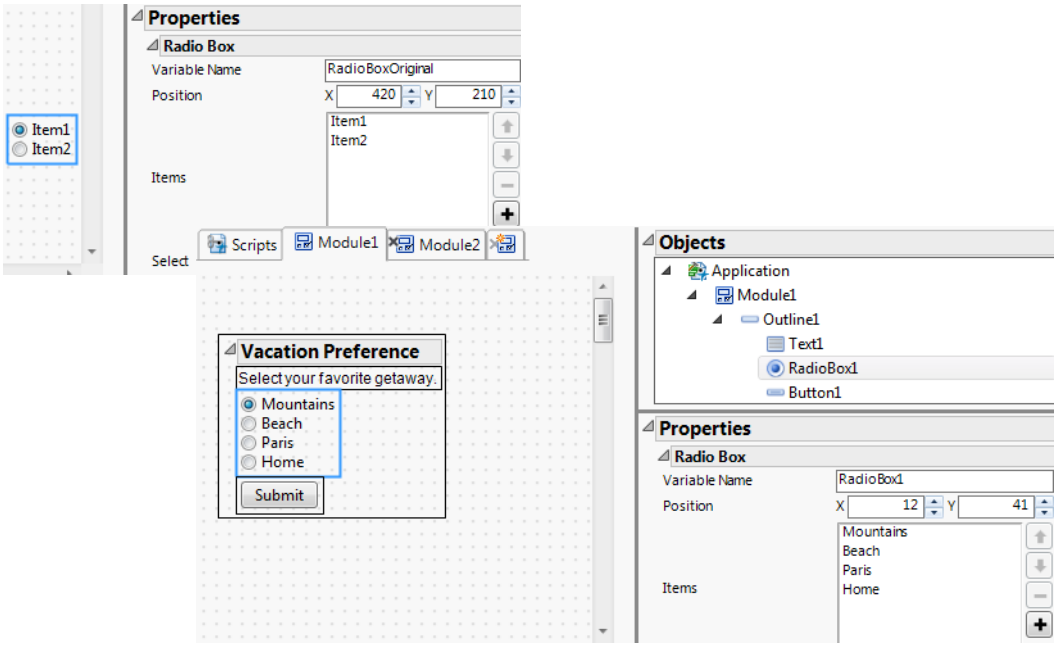
Customize Object Properties

After you drag an object from the Sources pane onto a module, customize the object properties in the Properties pane. 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-sensitive. When a script contains an object named "Button1", Application Builder warns if you try to rename another object "Button 1".

To create the check box list items in Figure 15.8, double-click the placeholder items "Item1" and "Item2" and enter the new list item.

Figure 15.8 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 pane, place your cursor 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.

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 260 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 pane. Then modify the following options in the Properties pane.

Variable Name Specifies the name of the data table object. This name appears in the Properties and Objects panes and in the application's JSL script.

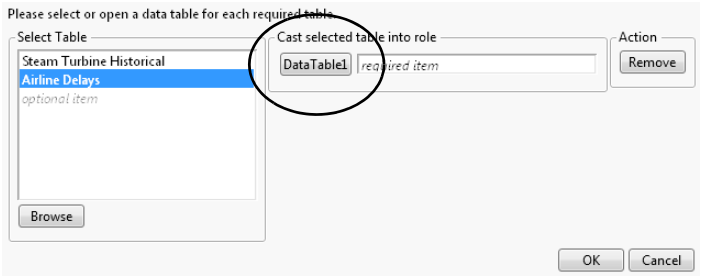
Path Specifies the absolute or relative filepath for the data table used in the application. You can precede the data table name with a path variable (such as $HOME or $USER_APPDATA).

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 15.9 shows the default value.

Figure 15.9 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.
- 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.
- 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.

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.

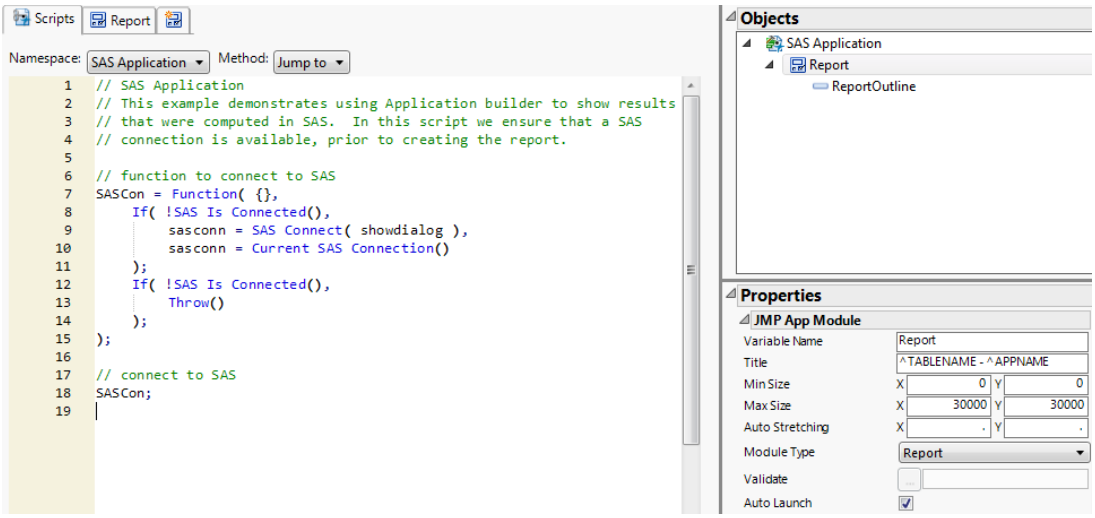
Application and Module Namespaces

To prevent variable names and values from conflicting among scripts, Application Builder automatically creates the Application namespace and a namespace for each module as follows:

- 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 pane). See Figure 15.10.

Figure 15.10 Application and Module Namespaces



There are two types of scripts: *named scripts* and *anonymous scripts*.

Named Scripts

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.

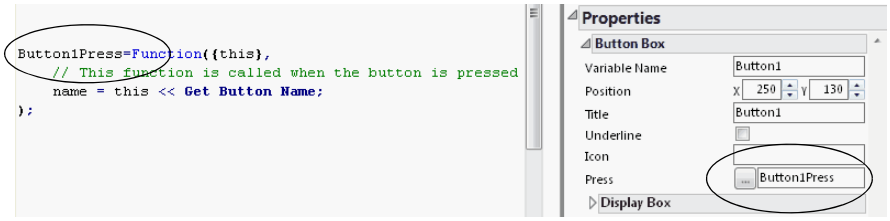
Two other advantages of named scripts: when you add a named script to an object, JMP inserts a placeholder script on the Scripts tab, which you then edit. On the Scripts tab, you can also select a script from the Method list to navigate to that code, which is particularly helpful in long scripts.

Add a named script as follows:

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 Macintosh, press OPTION, and then select **Scripts > Press**.)

   The object's placeholder script appears on the **Scripts** tab (Figure 15.11). The name of the named script, *Button1Press*, shows up in both the script and the object properties.

Figure 15.11 New Script and Script Properties



2. Edit the object's placeholder script and properties to provide the necessary functionality. For example, the radio box shown in Figure 15.11 reflects the following changes:
   - The Title property was changed to *Submit*, the text that appears on the button.
   - The `Close Window` function was added to the object. When the user clicks the **Submit** button, the window closes.

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 as follows:

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 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, select **Run Application** from the Application Builder red triangle menu.

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** (or **File > Export** on Macintosh), you can choose to save as an application source file (.jmpappsource), an application (.jmpapp), or a script (.jsl).

The **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.

Save Script to Data Table Lets you run the application from a data table script. This option is available only when the data table is open. When you edit this script from the data table, the application opens in Application Builder, not the script editor.

Save Script to Journal Embeds the application script in an open or new journal. Click the link to run the application.

Save Script to Script Window Lets you edit the script in an open or new 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.

Save Script to Add-In Lets JMP users install the application and launch it from a JMP menu. See "JMP Add-Ins" on page 646 for details about creating add-ins.

Additional Examples of Creating Applications

The following examples illustrate various uses for applications in JMP.

Parameterized Variables

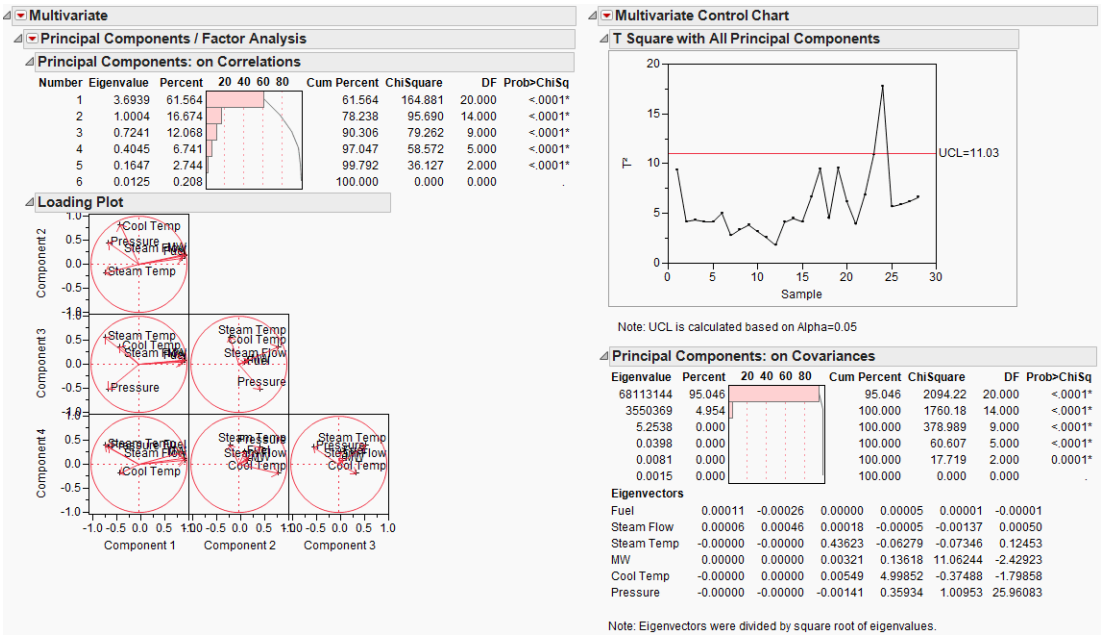
The following example shows how to create an application with parameterized variables. Users select the variables in a launch window, and then predefined reports are generated.

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**.

   The Application Builder window appears.
4. Enlarge the window.
5. In the Sources pane, drag each Multivariate report onto the application workspace in a single row.
6. Click both reports and then select **Format > Align Boxes > Align Top**.
7. Click each report and type `yvar` next to **Y Variable** in the Objects pane.
8. From the Application red triangle menu, select **Run Application**.
9. Select the Fuel, Steam Flow, and Steam Temp variables and then click the **Y** button.
10. Click **OK**.

    New Multivariate reports appear in one window (Figure 15.12).

Figure 15.12 Multivariate Application



Tip: The absolute path to the data table is inserted automatically in the application's Table property. You can use the $SAMPLE_DATA path variable instead or enter another absolute or relative path. Remember that the user must have access to this path.

Filtered 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 other reports contained in the same window.

To set up the filter, follow these steps:

1. Select the windows that you want to combine.
   - On Windows, select the check box in the lower right corner of the open data table or report window. Select **Combine selected windows** next to the check box.
   - On Macintosh, select **Window > Combine Windows**, select the open data table or report window, and then click **OK**.

   The objects appear in a new report window.
2. In the new window, select **Edit Application** from the red triangle menu.
3. In the Application Builder window, 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.

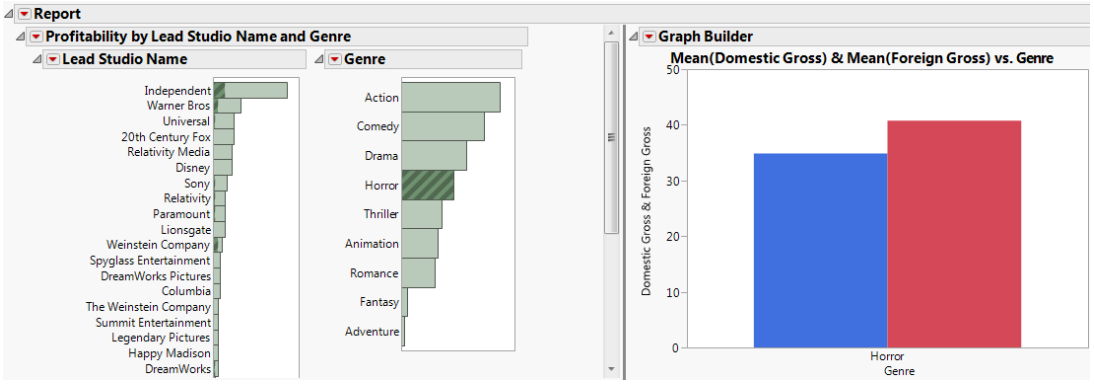
4. Select **Run Application** from the Application Builder red triangle menu.

   The reports appear in one window (Figure 15.13).

5. Test the application by selecting a variable in the primary report.

   Only data for the selected variable appears in the second report.

Figure 15.13 Example of Filtered Content



Date Selector

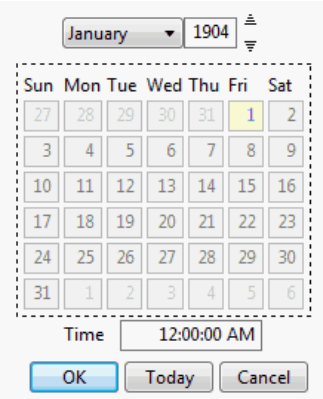
To insert a date selector window into an application, follow these steps:

1. Drag a Number Edit Box from the Sources pane to the workspace.
2. Select the Number Edit Box and then click the button next to Format in the Properties pane.
3. Select **Date** from the list and then select **m/d/y**.
4. Type 25 in the Width box.
5. Click **OK**.
6. Select **Run Application** from the Application Builder red triangle menu.

   The Number Edit Box, which shows a date, appears in a new window.

7. To open the date selector window, place the cursor over the box until a blue triangle appears.
8. Click the blue triangle to view the date selector window.

Figure 15.14 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.

JMP Add-Ins

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.

An add-in is essentially a collection of files zipped up into a single, saved file. You can share the add-in with other JMP users.

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 scripts.

Note: For more information about Application Builder, see the "Application Builder" on page 623.

Create an Add-In Using Add-In Builder

To create a JMP add-in:

- On Windows, select **File > New > Add-In**.
- On Macintosh, select **File > New > New Add-In**.

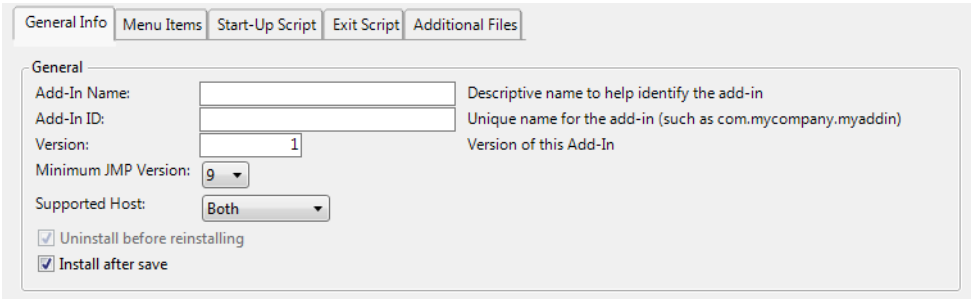
The process of creating an add-in involves the following steps:

- "Add General Information" on page 647

Add General Information

First, in the **General Info** tab, add the general information that identifies and sets up your add-in.

Figure 15.15 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.
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.

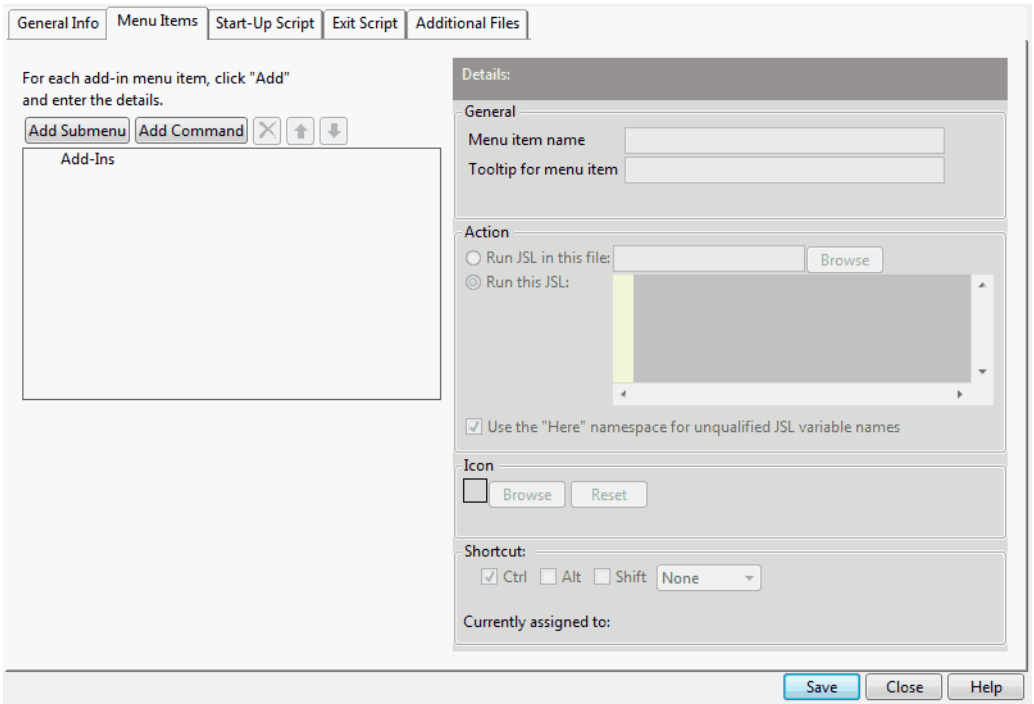
5. Select whether you want the add-in to be supported on Windows, Macintosh, or both.
6. (Optional) Select the check box next to **Install after save** if you want to install the add-in after saving it.

   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 15.16 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**, type the name of the submenu.

   This name appears in the **Add-Ins** menu.
4. Click **Add Command**.

5. Next to **Menu item name**, type 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 237 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**.

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 as follows:

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 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 Macintosh, 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.

Remove an Add-In from the Add-Ins Menu

To remove an add-in from the **Add-Ins** menu:

1. Select **View > Add-Ins**.
2. Select the add-in from the Registered Add-Ins list.
3. Deselect the **Enabled** check box.

Uninstall an Add-In

To uninstall an add-in:

1. Select **View > Add-Ins**.

2. Select the add-in from the Registered Add-Ins list.
3. Click **Unregister**.

Share an Add-In

Once you have a .jmpaddin file, you can share that with other users. E-mail 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)
  - %LOCALAPPDATA%\SAS\JMP\AddIns (only the current user on this machine can access the add-in)
- Add-In files on Macintosh 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.

Note the following:

- 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 Register Addin and Unregister Addin sections of the *JSL Syntax Reference* for details.
- For information about creating the addin.def file, see "Create an Add-In Manually" on page 652.

Note the following:

- 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:

`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.

`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.

`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.

`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).

`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.

`home_mac` Optional. The path to the add-in files to be used when JMP is running on the Macintosh. Overrides the value specified for home on Macintosh, if any.

`autoLoad` Optional, Boolean. The default value is True (1). Determines whether this add-in is initially configured to load automatically during JMP start-up.

`host` Optional. Valid values are `Win` and `Mac`.

`minJMPVersion` Optional. Valid values are integers corresponding to the JMP major version that is the minimum version that the add-in supports.

`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.

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\<version number>\Samples\Scripts
- On Macintosh: \Library\Application Support\JMP\<version number>\Sample\Scripts

Note: On Windows, in JMP Pro, the "JMP" folder is named "JMPPro". In JMP Shrinkwrap, the "JMP" folder is named "JMPSW".

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. 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/community/file-exchange.

Chapter 16

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 so you can copy and paste portions of code into your own scripts.

Contents

Run a Script at Start Up

If you want to run the same script every time you start JMP, name it 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". In JMP Shrinkwrap, the "JMP" folder is named "JMPSW".

On Windows:

1. C:\Users\<username>\AppData\Roaming\SAS\JMP\<version number>
2. C:\Users\<username>\AppData\Roaming\SAS\JMP

On Macintosh:

1. /Users/<username>/Library/Application Support/JMP/<version number>
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 searches for the administrator start-up script first, and runs it if found. Then JMP searches for the user start-up script, and runs it if found.

On Windows:

1. C:\ProgramData\SAS\JMP\<version number>
2. C:\ProgramData\SAS\JMP

On Macintosh:

1. /Library/Application Support/JMP/<version number>
2. /Library/Application Support/JMP

Convert Character Dates to Numeric Dates

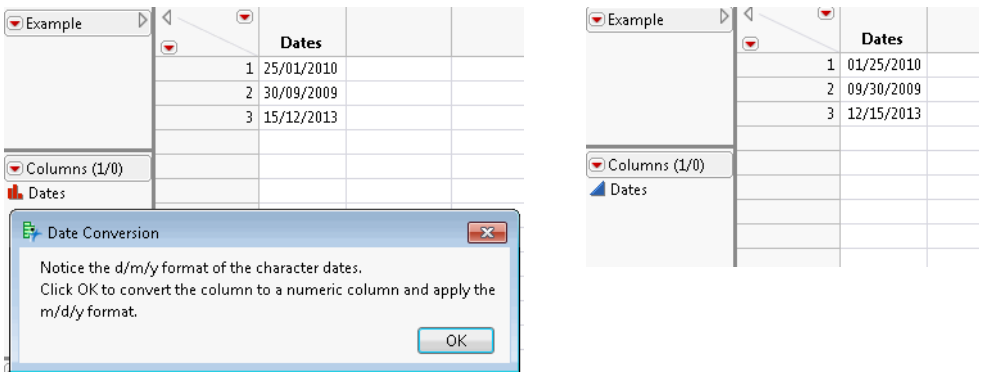
Data might appear to be numeric in the data table. However, the column properties may specify a character data type. To manipulate the data as date/time values, convert the column to a numeric column and specify how you want the values to appear.

Convert Dates.jsl (Figure 16.1) 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 16.2).

Figure 16.1 Script for Converting Character Dates to Numeric Dates

```
/* This script demonstrates how to convert a column of character dates to numeric dates. */

// 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" );

// Display the data table in front of the script.
dt << Data Table Window();
```

Figure 16.2 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

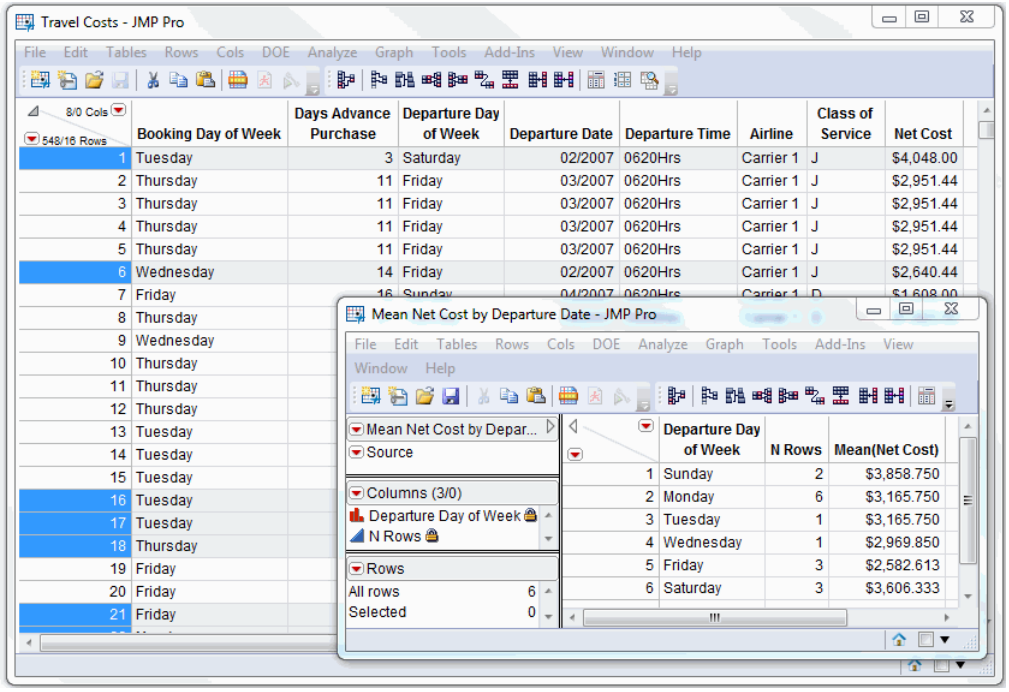
How can you work with dates in JMP? JMP provides a number of formats that you can use to make comparisons and then subset data based on the date.

Select Where Using Dates.jsl (Figure 16.3) 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 16.4).

Figure 16.3 Script for Selecting Dates

```
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 = nt << Summary(
    Group( :Departure Day of Week ),
    Mean( :Net Cost ),
    Output Table Name( "Mean Net Cost by Departure Date" )
);
```

Figure 16.4 The Original Table and the Final Summary Table



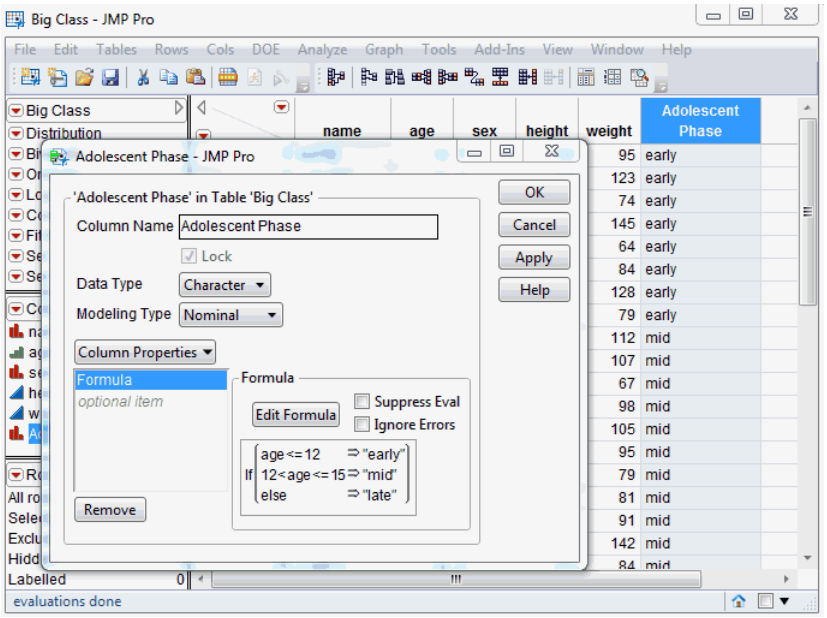
Create a Formula Column

How do you create a formula column that combines conditional expressions with value comparisons? Create a Formula Column.jsl (Figure 16.5) shows how to create a new formula column that evaluates ages in Big Class.jmp and returns the result in the new column (Figure 16.6).

Figure 16.5 Script for Creating a Formula 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 16.6 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 (Figure 16.7 and Figure 16.8) and performs a Bivariate analysis and shows results such as the sample size, RSquare, and Correlation in a new report window. Figure 16.9 shows the Bivariate report along with the customized report, though the Bivariate report window closes after the script is finished.

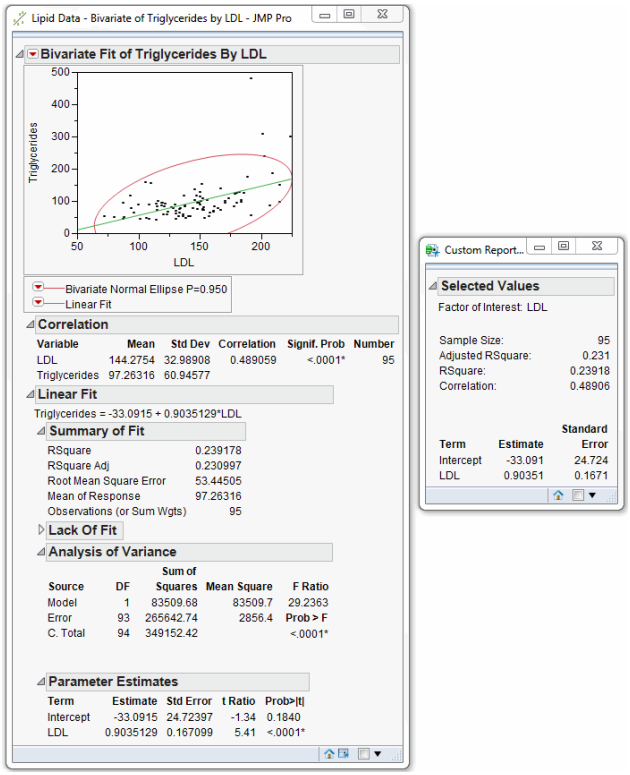
Figure 16.7 Script for Extracting Values into a Custom Report (Part 1)

```
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();
```

Figure 16.8 Script for Extracting Values into a Custom Report (Part 2)

```
// 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 ), ),
        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 )
        )
    )
);

Close( sd ); // Close the data table.
```

Figure 16.9 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 (Figure 16.10 and Figure 16.11) asks the user to enter a number and then factors the number or confirms it as a prime number (Figure 16.12). This script is a good example of aligning several types of display boxes, concatenating text, and working with conditional functions.

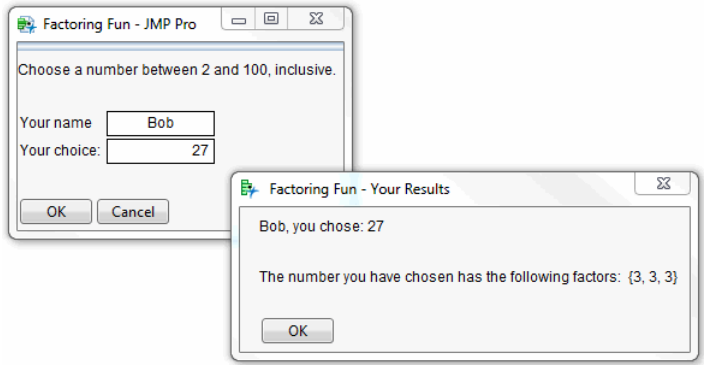
Figure 16.10 Script for an Interactive Program (Part 1)

```
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 reponses.
                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.
```

Figure 16.11 Script for an Interactive Program (Part 2)

```
                fromuser1 = fromuser0;   // Make a copy of the value for processing.
                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 ),
                        Insert Into( factors, fromUser0 ); // If found, place value in factor list.
                        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

                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 16.12 Factor Numbers Interactively



Appendix A

Compatibility Notes

Changes in Scripting from JMP 11 to JMP 12

The new General preference “Show log warnings for JSL compatibility changes in JMP 12” is selected by default to print compatibility warnings to the log.

Compatibility Issues

Evaluating Variable Names as Arguments

Message arguments are now evaluated as a rule. Previously, some messages would interpret a variable name as a string value. For example, the following script results in a linear scale in JMP 12 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 new “Allow Unquoted Strings in JSL” General preference to “Yes (with a warning)” or “No”.

Missing Value Codes

Column properties such as Missing Value Codes assign data values that produce incorrect calculations in `Col` functions (such as `Col Mean()`). To use the value in the data table cell instead of the missing value, refer to `Col Stored Value()` in the formula:

```
Mean( Col Stored Value( :x1 ), :x2, :x3 )
```

Missing Value Codes are correctly recognized in other functions.

Button Boxes Names and Line Breaks

Icons appear on button boxes that have empty names.

Line-break characters are ignored in button boxes.

Competing Cause Summary Report

The Competing Cause Summary report provides more details about parameter estimates. Scripts that get information from the Cause Summary table might need to be modified.

Confidence Intervals in Proportional Hazard and Parametric Survival

In the Proportional Hazard and Parametric Survival platforms, the Likelihood Ratio Tests and Likelihood Confidence Intervals have been separated. Scripts that get confidence intervals out of the report might need to be modified.

Coordinates Returned for Log Scale Axes

`XOrigin()`, `YOrigin()`, `XRange()`, and `YRange()` return the original coordinates when the axis is set to log scale. Previously, log-transformed values were returned.

Evaluating Boolean Arguments

Messages with empty Boolean arguments are more consistently evaluated. For all messages except row state messages, omitting the argument enables the option.

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

The keywords `"true"`, `"yes"`, `"present"`, `"on"`, and `1` consistently enable the option:

```
obj << Unequal Variances( "true" );
obj << Unequal Variances( "yes" );
obj << Unequal Variances( "present" );
obj << Unequal Variances( "on" );
obj << Unequal Variances( 1 );
```

Similarly, `"false"`, `"no"`, `"absent"`, `"off"`, and `0` disable an option.

To toggle these options, use the `"switch"`, `"toggle"`, or `"flip"` argument.

For row state messages, an empty argument always toggles the option. If the option is off, the message turns it on. If the option is on, the message turns it off.

Font Properties

An empty `obj<<Set Font` message now displays the operating system window for choosing a font.

<<Get Script with an Excluded Column

When you run `<<Get Script` on a data table that has an excluded column, the returned script includes the `Exclude` argument. In JMP 11, `Name( "Exclude/Unexclude" )` was returned in the script.

JMP Product Name()

`JMP Product Name()` returns “Student” in JMP Student Edition.

Microsoft Excel Preference for Opening Individual Worksheets

`Excel Selection(1)` is an alias for the `Excel Open Method( "Select Individual Worksheets(1)" )` preference. If the JMP preference file contains an Excel Open Method, that preference takes precedence over the script.

Non-Recursive Files in Directory() Returns Directory Names on Macintosh

The non-recursive `Files in Directory()` returns directory names on Macintosh to match the Windows functionality. Now you must test for directory names when using `Files in Directory()` non-recursively on both Windows and Macintosh.

On Open() Scripts

`On Open()` scripts prompt you to run the script instead of running automatically. A new Tables preference called "Evaluate OnOpen Scripts" controls this behavior.

Parametric Survival()

`Residual Quantile Plot()` is now named `Residual Plot()`.

Random Seed in DOE

Existing scripts that reproduce a design using a random seed do not reproduce the design in JMP 12. You must specify the random seed *and* the number of starts that were used when you originally wrote the script.

Reliability Forecast Scripts

The "Sequential" forecast type has been renamed "Incremental". Existing scripts still run. However, you must update the text in reports and data tables if scripts refer to it.

Run Program() on Macintosh

In `Run Program()` on Macintosh, the full path to the program must be specified to locate the executable. Type the following command in Terminal to determine the full path to a program:

```
whereis <program>
```

Deprecated JSL

Please begin using the new following new JSL commands for rotating axis labels and setting fonts. Existing scripts continue to work. However, the old messages and syntax will be discontinued in the future.

Rotated Labels

`The <<Rotated Labels` message for `Axis Box()` has been replaced with `<<Label Orientation`.

```
ab<<Label Orientation( "Automatic" );
ab<<Label Orientation( "Horizontal" );
ab<<Label Orientation( "Vertical" );
ab<<Label Orientation( "Perpendicular" );
ab<<Label Orientation( "Parallel" );
ab<<Label Orientation( "Angled" );
```

Set Font() Arguments

The following formats are deprecated for ContourSeg(), TextSeg(), HierBox(), ColListBox(), FilterColSelector(), ListBoxBox(), NumberEditBox(), OutlineBox(), TextBox(), and TextEditBox():

```
obj<<Set Font( {"Name", size, style, angle} );
obj<<Set Font( Face( "Name" ), Size( size ), Style( style ) );
```

Use the following formats, which work for all display boxes that have font properties:

```
obj<<Set Font( "Name", size, style, angle );
obj<<Set Font Name( "Name" ) << Set Font Size( size ) << Set Font Style( style
   );
```

Appendix B

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 operator, 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 is a yes/no value, something that is on or off, shown or hidden, true or false, 1 or 0, yes or no. An operator listed as *being a Boolean operator* 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. This book prefers the more specific terms **operator**, **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.

infix operator In 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` operator. 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` operator.

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 521 and "MouseTrap" on page 524.

mouseup An event generated by releasing the mouse button. See "Handle" on page 521 and "MouseTrap" on page 524.

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, `title("My Histogram")` in the `Graph Box` function is a named argument. In functions such as `New Window`, the `title` is *not* a named argument, because `title` is the first required 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.

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.

Index

Scripting Guide

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W-Z