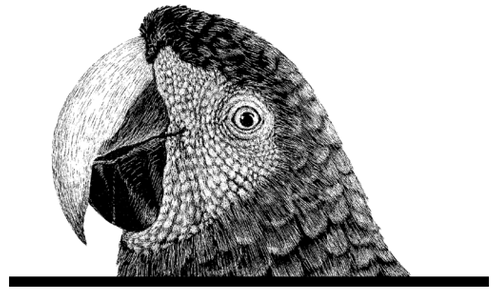


LINGO
IN A NUTSHELL

A Desktop Quick Reference

Bruce A. Epstein

O'REILLY™

Cambridge • Köln • Paris • Sebastopol • Tokyo

Lingo in a Nutshell
by Bruce A. Epstein

Cover illustration by Susan Hart, Copyright © 1998 O'Reilly & Associates, Inc.
Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Tim O'Reilly

Production Editor: Paula Carroll

Editorial and Production Services: Benchmark Productions, Inc.

Printing History:

November 1998: First Edition.

This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

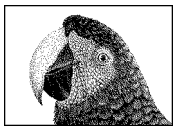
ISBN: 1-56592-493-2

Table of Contents

Part II: Lingo Data Types and Expressions

Part III: Lingo Events

Part VI: Appendixes



Preface

About This Book

You are holding in your hands one half of *Bruce's Brain in a Book*. The other half of my brain is in the companion book, *Director in a Nutshell*. These books are the distillation of years of real-life experience with countless Director projects plus many hours spent researching and testing Director 6's and 6.5's new features. While they can be used separately, they are ideally used as a single two-volume reference, which can be purchased together for less than most single Director books.

Lingo in a Nutshell focuses on the abstract concepts in Lingo—variables, scripts, Behaviors, objects, mouse and keyboard events, timers, math, lists, strings, and file I/O. *Director in a Nutshell* focuses on the concrete aspects of Director—the Cast, the Score, Projectors, MIAWs, media (graphics, sound, digital video, text), Director's windows, GUI components (buttons, cursors, menus) and Shockwave.

If you already know a lot about Director or have been disappointed by the existing documentation, these are the books you've been waiting for. They address many of the errors and omissions in Macromedia's documentation and in many third-party books. There is no fluff or filler here, so you'll miss a lot if you skim. We are both busy, so let's get on with it.

What Are These Books and Who Are They for?

Director in a Nutshell and *Lingo in a Nutshell* are desktop quick references for Director and Lingo developers who are familiar with Director's basic operation and need to create, debug, and optimize cross-platform Director and Shockwave

projects. These books are concise, detailed, respectful of the reader's intelligence, and organized by topic to allow quick access to thorough coverage of all relevant information.

Because Lingo and Director are inextricably linked, I have kept all information on a single topic within a single chapter rather than breaking it along the traditional Director versus Lingo lines (with the exception of the *Using Xtras* and *Lingo Xtras and XObjects* chapters). Don't be fooled by the titles of the books; both include a lot of Lingo, and they should be read in parallel.

Director in a Nutshell should not be confused with the third-party books that merely rehash the manuals, nor should it be considered an introductory book. It is exceptionally valuable for non-Lingo users but also covers Lingo related to those aspects of Director mentioned previously. *Lingo in a Nutshell* covers both the very basics of Lingo and its most advanced features. It is for both new and experienced Lingo programmers, which may sound impossible but isn't. Each book covers both Windows and the Macintosh.

To describe them as "beginner," "intermediate," or "advanced" would be misleading because they cover both the very basic foundation of Director and its very advanced usage. Strictly as a comparison to other books on the market, you should consider their *coverage* extremely advanced, but the text itself is accessible to Director users of all levels. *Lingo in a Nutshell* allows Director users to take full advantage of Lingo's power, and *Director in a Nutshell* helps users of all levels deal confidently with the spectrum of Director's media types and features.

What These Books Are Not

These books are not a rehash of the Director manuals or Help system but rather a complement to them; as such, they are unlike any other books on the market.

These books are not a celebration of Director as multimedia Nirvana. They are for people who know that Director has many quirks and some bugs and want to know how to work around them quickly and effectively.

These books are not courses in graphic design, project management, Photoshop, HTML, or JavaScript. They will, however, help you to integrate your existing skills and external content into Director's framework.

These books are not a Director tutorial; I assume that you are familiar with the basics of Director's Cast, Score, Stage, and menus. They are not for people who need hand-holding. They are for people who can apply general concepts to their specific problem and want to do so rapidly.

These books are not perfect—errors are inevitable—so use them as a guide, not the gospel. Although these books cannot anticipate all circumstances, they do provide the tools for you to confidently solve your specific problems, even in the face of erroneous or incomplete information.

Last, these books are not a static lecture. They are an ongoing conversation between you, the reader, and me, the author. Feedback from many customers, clients, and friends has already shaped their content. I have packed them with facts, but I also provide the tools to allow you to understand and debug any situation. Let us see if

we can solve some problems in Director and learn something about ourselves along the way.

Lingo in a Nutshell

Lingo in a Nutshell covers the abstract aspects of Lingo that exist apart from its control over media elements, the Cast, and the Score. This book covers the spectrum from Lingo basics to advanced scripting with Lists, Behaviors, and Xtras. It is divided into five major sections.

Part I, *Lingo Basics*

Chapter 1, *How Lingo Thinks*, defines the Lingo language and its syntax including handlers, variables, and Lingo control structures. Refer also to Chapter 1, *How Director Thinks* in *Director in a Nutshell*.

Chapter 2, *Events, Messages, and Scripts*, explains where, when, and how to use various types of scripts to detect user and system events. It covers the new event and message passing in Director 6, including details on trapping events with Behaviors.

Chapter 3, *Lingo Coding and Debugging Tips*, helps you plan your Lingo and covers the Debugger, Message window, and Lingo error messages. See also Appendix E, *Error Messages and Error Codes*.

Chapter 4, *Lingo Internals*, is designed for experienced programmers and compares Lingo's syntax, commands, and structure to those of other languages. Refer also to the downloadable Chapter 20, *Lingo for C Programmers*.

Part II, *Lingo Data Types and Expressions*

Chapter 5, *Data Types and Expressions*, covers Lingo data types, implicit and explicit type conversion and coercion, type checking, logical expressions, comparison operators, and string operators.

Chapter 6, *Lists*, is a life-saving chapter covering the commands for linear lists, property lists, points, and rectangles in well-organized tables. It contains numerous examples including randomized and non-repeating lists.

Chapter 7, *Strings*, covers string expressions, concatenation, parsing, and manipulation, and chunk expressions (characters, words, items, lines, and fields). See also Chapter 12, *Text and Fields*, in *Director in a Nutshell*.

Chapter 8, *Math (and Gambling)*, covers arithmetic operators, math functions, exponentiation, geometry and trig functions, rounding and truncation, and random numbers.

Part III, *Lingo Events*

Chapter 9, *Mouse Events*, covers responding to mouse clicks and cursor movement, including how to make buttons with rollover and highlight states.

Chapter 10, *Keyboard Events*, covers responding to keyboard input and trapping various key combinations (including modifier keys, function keys, arrow keys, and the numeric keypad).

Chapter 11, *Timers and Dates*, covers timers, time-outs, dates, times, and unit conversion.

Part IV, *Applied Lingo*

Chapter 12, *Behaviors and Parent Scripts*, helps you make the most of Behaviors and other object-oriented scripting techniques.

Chapter 13, *Lingo Xtras and XObjects*, covers using Xtras and XObjects to extend Lingo's scripting language. See also Chapter 10, *Using Xtras*, in *Director in a Nutshell*.

Chapter 14, *External Files*, covers the *FileIO* Xtra for reading and writing files from within Director. It also covers commands that work with external Director-related files and non-Director documents and applications.

Chapter 15, *The MUI Dialog Xtra*, covers using the *MUI* Xtra to create basic Alert dialogs. Refer also to the downloadable Chapter 21, *Custom MUI Dialogs*, which provides painstaking detail on creating custom dialogs with the *MUI* Xtra.

Part V, *Lingo Command Reference*

Chapter 16, *Enumerated Values*, lists all the Lingo commands that accept or return numeric codes, symbols, or strings that indicate particular values, including transitions, ink effects, palettes, cursors, and window types.

Chapter 17, *Changed, Undocumented, and Misdocumented Lingo*, covers Lingo commands that are undocumented or misdocumented or behave differently in Director 6 than in prior versions of Director.

Chapter 18, *Lingo Keyword and Command Summary*, is a complete list of every command, function, symbol, and string recognized by Lingo, including a syntax example.

Chapter 19, *The Lingo Symbol Table*, explains the inner workings of the hidden Symbol Table and symbols in general. Refer also to the downloadable Chapter 22, *Symbol Table Archaeology*, for additional details.

Part VI, *Appendixes*

Appendix A, *ASCII Codes and Key Codes*

Appendix B, *Changes in D6 Through D6.5*

Appendix C, *Case-Sensitivity, Sort Order, Diacritical Marks, and Space-Sensitivity*

Appendix D, *The DIRECTOR.INI and LINGO.INI Files*

Appendix E, *Error Messages and Error Codes*

Glossary

The economics of print publishing precluded me from including everything in this book. The good news is that the material (plus many more examples) is available online in PDF (Acrobat) format (see *http://www.zeusprod.com/nutshell*).

Online Bonus Chapters:

Chapter 20, *Lingo for C Programmers*, is designed for experienced programmers and compares Lingo's syntax, commands, and structure to C. It picks up where Chapter 4, leaves off.

Chapter 21, *Custom MUI Dialogs*, covers the excruciating details of using the MUI Xtra to create custom dialog boxes. It expands on Chapter 15, which covers only the MUI Xtra's *Alert()* method.

Chapter 22, *Symbol Table Archaeology*, covers the history and hidden secrets of the Lingo Symbol Table and complements Chapter 19.

The companion volume, *Director in a Nutshell*, covers content development and delivery in Director. It also covers media and user interface elements and the Lingo to control them. Refer to the Preface in *Director in a Nutshell* for details.

Conventions Used in This Book

Typographical Conventions

- Lingo *keywords* (*functions*, *commands*, and *property names*) are shown in *italics* except in tables where they are italicized only when necessary to distinguish them from the surrounding text. Italics in tables usually indicate replaceable values.
- *`Arguments`*, *`user-specified`*, and *`replaceable`* items are shown in *`italics`* and should be replaced by real values when used in your code.
- New terms are shown in *italics* and are often introduced by merely using them in context. Refer to the Glossary for details.
- Menu commands are shown as `MenuName➤MenuItem`.
- Options in dialog boxes, such as the *Tab to Next Field* checkbox, are shown in *italics*.
- Constants, such as `TRUE`, `FALSE`, and `RETURN`, are shown in `Courier`.
- `#symbols` are preceded by the # character and shown in `Courier`.
- Optional items are specified with curly braces ({}) instead of the traditional square braces ([]) which Lingo uses for lists. For example:

  ```
  go {to} {frame} whichFrame
  ```

  means that the following all are equivalent:

  ```
  go whichFrame
  go to whichFrame
  go to frame whichFrame
  go frame whichFrame
  ```

- Allowed values for a property are separated by a "|". The following indicates that *the alignment of member* property can be set to "left," "right," or "center":

  ```
  set the alignment of member 1 = "left" | "right" | "center"
  ```

Grammatical and Stylistic Conventions

- Most Lingo properties start with the word *the*, which can lead to sentences such as "The *the member of sprite property* can be changed at runtime." I often omit the keyword *the* preceding properties to make sentences or tables more readable, but you should include the *the* in your Lingo code.

- Lingo event handlers all begin with the word *on*, such as *on mouseUp*. I often omit the word *on* when discussing events, messages, and handlers or in tables where the meaning is implied.
- Be aware that some Director keywords are used in multiple contexts such as the *on mouseUp* event handler and the *the mouseUp* system property. The intended usage is discernible from context, or stated explicitly in ambiguous circumstances.
- I use terminology fairly loosely, as is typical among Lingo developers. For example, a "*mouseUp* script" is technically "an *on mouseUp* handler within a script." The meaning should be clear from the context.
- I capitalize the names of Director entities, such as the Score, the Stage, the Cast, and the Message window. I don't capitalize general terms that refer to classes of items, such as sprite scripts.
- Most handler names used in the examples are arbitrary, although handlers such as *on mouseUp* that trap built-in events must be named as shown. I use variable names like `myThing` or `whichSprite` to indicate items for which you should substitute your own values. When in doubt, consult Table 18-1, "*Lingo Command and Keyword Summary*.
- I use few segues and assume you will reread the material until it makes sense. As with a Dalí painting, you must revisit the text periodically to discover details that you missed the first time.

Examples

- Example code is shown monospaced and set off in its own paragraph. If a code fragment is shown, especially using the `put` command, it is implicit that you should type the example in the Message window to see the result. Any text following "`--`" is the output from Director or a comment from me:

```
set x = 5    -- Set the variable x to 5
put x        -- Display the value of x
-- 5
```

- Long lines of Lingo code are continued on the next line using the Lingo continuation character ([LC]) as shown here. This character is created using `Opt-Return` or `Option-L` (Macintosh) or `Alt-Enter` (Windows).

```
set the member of sprite (the currentSpriteNum) = ¬
   member "Hilighted Button"
```

- If you have trouble with an example, check for lines that may have been erroneously split without the Lingo continuation character (¬). Remember to use parentheses when calling any function that returns a value. Otherwise you'll either see no result or receive an error.

```
rollover         -- wrong
rollover()       -- wrong
put rollover     -- wrong
put rollover()   -- correct
```

- I sometimes use the single-line form of the *if...then* statement in an example for brevity. You should use multiline *if...then* statements in your code. See Chapter 1 for details on the *if* statement.

```
-- This will usually work
if (x > 5) then put "It's True!"
-- But this is more reliable
if (x > 5) then
   put "It's True!"
end if
```

- If a handler is shown in an example, it is implied that the handler has been entered into the appropriate type of script. Generally, mouse event handlers such as *mouseUp* belong in sprite scripts; frame events handlers such as *exitFrame* belong in frame scripts; and custom utilities belong in movie scripts. I often show a handler followed by an example of its use. Type the handler into an appropriate script, and then test it from the Message window. If I don't show a test in the Message window, either the handler does not output a visible result, or it is assumed that you will test it yourself.

```
-- This goes in a script, in this case a movie script
on customHandler
  put "Hello Sailor!"
end customHandler

-- This is a test in the Message window
customHandler
-- "Hello Sailor"
```

- The output shown may vary inconsequentially from the results you would see based on your system setup. Most notably, the number of decimal places output for floating-point values depends on your setting for *the floatPrecision* property.
- If the output of a handler is extremely long, the results will not be shown in their entirety or may not be shown at all.
- The examples are demonstrative and not necessarily robust, and they assume that you provide valid inputs when applicable. It is good practice to include type checking and error checking in your actual Lingo code, as described in Chapters 1 and 3. I often omit such checking to keep examples shorter and to focus on the main issue at hand.
- Some examples, particularly the tests performed from the Message window, are code *fragments* and won't work without help from the studio audience. You should ensure that any variables required by the examples (particularly lists) have been initialized with meaningful values, although such initialization is not shown. For example:

  ```
  put count (myList)
  ```

  The previous code fragment assumes that you have *previously* set a valid value for *myList*, such as:

  ```
  set myList = [1, 7, 5, 9]
  ```

- Some examples allude to field cast members, such as:

```
set the text of field "Memory" = string(the freeBlock)
```

It is implied that you should create a field cast member of the specified name in order for the example to work.

- Screen shots may not match your platform's graphical user interface exactly.
- I present a simplified view of the universe whenever my assumptions are overwhelmingly likely to be valid. You can intentionally confuse Director by setting bizarre values for a property or performing malicious operations, such as deleting elements from a *rect* structure, but you do so at your own risk. I cover situations where errors might occur accidentally, but you should assume that all statements presented as fact are prefaced by, "Assuming you are not trying to screw with Director just for fun..." When they are likely to be relevant, I state my assumptions clearly.
- The myriad ways to perform a given task are shown when that task is the main topic of discussion but not if it is peripheral to the subject at hand. When it is incidental, I may show the most expedient or clearest method rather than the suggested method.
- Examples are usually self-contained, but they may rely on custom handlers shown nearby. If an example builds on previous examples or material cross-referenced in another chapter, it is assumed that the relevant handlers have been entered in an appropriate script (usually a movie script).
- What rightly belongs in one table sometimes is broken into two or three due to space constraints. Similar information may be organized in different ways in multiple tables to help you find what you want, especially in Chapter 6. The first column of each table contains the table's "key item" for which full details are provided. Subservient items, for which the table may not contain complete information, are relegated to other columns. For example, if a function is listed in the "*See Also*" column of a table, complete details on that command can be found in surrounding prose or other tables.

Refer to the Glossary for a complete list of definitions.

New Features in Director 6 and 6.5

Score, Sprites, Auto-Puppeting, and Paths

Director's new Score is *radically* different and includes a Sprite Toolbar and customizable views. Sprites receive several new messages (*beginSprite*, *endSprite*, *mouseEnter*, *mouseLeave*, etc.), allowing them to be managed much more easily. Refer to Chapter 3, *The Score and Animation*, in *Director in a Nutshell* and to Chapter 2 in *Lingo in a Nutshell*.

Help and Manuals

The new Help system includes a *lot* of information that is not in the manuals, plus many useful *Show Me* demonstration movies. Choose *Show Me* from the Help menu or from the Help *Contents* window for demonstrations of many of Director's new features.

New Behaviors, Messages, Cue Points, and Lingo

Behaviors allow you to easily add *multiple* scripts to a sprite. Director's message passing has been radically revised, and there are many new

messages, including rollover events and error trapping. Refer to Chapter 2, 9, and 12. Director now supports cue points for synchronizing sounds with animation in the Score (see Chapter 15, *Sound and Cue Points*, in *Director in a Nutshell*).

Shockwave and Internet Improvements
: Shockwave and Director now support streaming playback of Internet-based content. Many Director commands support linking to a URL, and linked cast libraries or streaming Shockwave audio can reside on the Internet. New commands (*frameReady*, *mediaReady*, *netDone*, etc.) support asynchronous operations in Director via the *NetLingo* Xtra. Refer to Chapter 11, *Shockwave and the Internet*, in *Director in a Nutshell*.

  Shockwave Audio is now integrated with Director. Local sounds can be compressed as well as those on the Internet. See Chapter 15 in *Director in a Nutshell*. Shockwave-style movie compression is also available to make your local Director project use less disk space.

New Media Formats and Application Integration
: Director 6.0 supports many new media formats.

Using external media (including sounds) at runtime requires the *MIX Services* Xtra plus the support Xtra for that particular file type, such as *Sound Import Export*.

Refer to Chapter 4, *CastLibs, Cast Members and Sprites*, and Chapter 10, *Using Xtras*, in *Director in a Nutshell* for details on file import types and the required Xtras.

New Features in Director 6.5
: Director 6.5 is the same as version 6.0.2 with the addition of many Xtras. See *http://www.macromedia.com/software/director/productinfo/newfeatures/* and refer to Appendix B, *Changes in D6 Through D6.5*. Refer also to *Director in a Nutshell*, especially to Appendix B, *New Features in Director 6.5*, and Chapter 16, *Digital Video*, which covers QuickTime 3.

Director Resources

The best thing about Director is the extended community of developers that you can torment for assistance. This book notwithstanding, Director is largely undocumented. Visit Macromedia's web site frequently, and plug into the broader Director community via mailing lists and newsgroups.

Director in a Nutshell and Lingo in a Nutshell

O'Reilly and Associates
: *http://www.oreilly.com/catalog/directnut/*

  http://www.oreilly.com/catalog/lingonut/

Home page for both books
http://www.zeusprod.com/nutshell

Download example code
http://www.zeusprod.com/nutshell/examples.html

Downloadable bonus chapters (PDF format)
http://www.zeusprod.com/nutshell/chapters.html

Links page (all URLs in this book are available by chapter/topic)
http://www.zeusprod.com/nutshell/links.html

Web Review—All things browser and web related
http://www.webreview.com/

Macromedia

Macromedia home page and mirror sites
http://www.macromedia.com

http://www-euro.macromedia.com

http://www-asia.macromedia.com

Director 6.5 update
http://www.macromedia.com/software/director/productinfo/newfeatures/

http://www.macromedia.com/software/director/upgrade/

Director Developers Center (searchable database of Tech Notes and tips)
http://www.macromedia.com/support/director/

http://www.macromedia.com/support/search/

http://www.macromedia.com/support/director/how/subjects/

Shockwave Developer Center
http://www.macromedia.com/shockwave/

http://www.macromedia.com/support/director/how/shock/

Dynamic HTML and Shockwave browser scripting
http://www.dhtmlzone.com/swdhtml/index.html

Director-related newsgroups
http://www.macromedia.com/support/director/interact/newsgroups/

news://forums.macromedia.com/macromedia.plug-ins

news://forums.macromedia.com/macromedia.director.basics

news://forums.macromedia.com/macromedia.director.lingo

Priority Access (fee-based) technical support
http://www.macromedia.com/support/techsupport.html

http://www.macromedia.com/support/director/suprog/

Beta program
http://www.macromedia.com/support/program/beta.html

Director feature suggestions
mailto:wish-director@macromedia.com

Phone support

MacroFacts (fax information) 1-800-449-3329 or 1-415-863-4409

Technical Support 1-415-252-9080

Main Operator: 1-415-252-2000

User groups

http://www.macromedia.com/support/programs/usergroups/worldwide.html

Developer Locator (find a Director or Lingo developer in your area)

http://www.macromedia.com/support/developer_locator/

Online services

CompuServe: Go Macromedia

AOL: The Macromedia forum on AOL no longer exists.

Macromedia User Conference (UCON), May 25–27 1999, in San Francisco, CA (There will be no UCON in the fall of 1998.)

http://www.macromedia.com/events/ucon99/

Web Sites and Xtras

Zeus Productions (my company) technical notes and Xtras

http://www.zeusprod.com

UpdateStage—monthly technical articles and the Director Quirk List and Xtras

http://www.updatestage.com

ftp://ftp.shore.net/members/update/

Director Online Users Group (DOUG)—articles, interviews, reviews

http://www.director-online.com

Maricopa Director Web—the mothership of Director information

http://www.mcli.dist.maricopa.edu/director/tips.html

ftp://ftp.maricopa.edu/pub/mcli/director

Lingo Behavior Database (example Behaviors) maintained by Renfield Kuroda

http://www.behaviors.com/lbd/

Peter Small's Avatars and Lingo Sourcery (far-out stuff)

http://avatarnets.com

Links to additional third-party web sites

http://www.mcli.dist.maricopa.edu/director/net.html

http://www.macromedia.com/support/director/ts/documents/tn3104-dirweb-sites.html

Third-Party Xtras

http://www.macromedia.com/software/xtras/director

FMA Online (Links to many Xtra developers)

http://www.fmaonline.com

Xtras developer programs

http://www.macromedia.com/support/program/xtrasdev.html

http://www.macromedia.com/support/xtras.html

QuickTime
http://quicktime.apple.com/

Microsoft
http://support.microsoft.com/

Mailing Lists

If you have the bandwidth, these mailing lists are often useful resources for Director, Shockwave, Xtras, and Lingo questions (see the Macromedia newsgroups). These mailing lists generate a *lot* of mail. Subscribe using DIGEST mode to avoid hundreds of separate e-mails each day.

DIRECT-L (Director and Lingo)
Archives: *http://www.mcli.dist.maricopa.edu/director/digest/index.html*

MailList: *http://www.mcli.dist.maricopa.edu/director/direct-l/index.html*

Send the following in the *body* of an e-mail to *listserv@uafsysb.uark.edu*:

```
SUBSCRIBE DIRECT-L yourFirstName yourLastName
SET DIRECT-L DIGEST
```

Lingo-L (Lingo)
http://www.penworks.com/LUJ/lingo-l.cgi

ShockeR (Shockwave)
Archive: *http://ww2.narrative.com/shocker.nsf*

MailList: *http://www.shocker.com/shocker/digests/index.html*

Send the following in the *body* of an e-mail to *list-manager@shocker.com:*

```
SUBSCRIBE shockwave-DIGEST yourEmail@yourDomain
```

Xtras-L (Xtras for Director)
http://www.gmatter.com/xtras-l.html

Send the following in the *body* of an e-mail to *listserv@gmatter.com:*

```
SUB XTRAS-L yourFirstName yourLastName
```

Dedication

Lingo in a Nutshell is dedicated to my wife, Michele, whose love makes my life worthwhile.

Acknowledgments

I am indebted to many people, some of whom I've undoubtedly omitted from the list below. Please buy this book and recommend it to friends so that I can thank the people I've forgotten in the next revision.

My deep appreciation goes out to the entire staff at O'Reilly, whose patience, professionalism, and unwavering dedication to quality are directly responsible for bringing these books to market. Special thanks go to my editors, Tim O'Reilly, Katie Gardner, and Troy Mott, and to Edie Freedman, Sheryl Avruch, Frank Willison, Nancy Priest, Rob Romano, Mike Sierra, Paula Carroll, Nancy Kruse

Hannigan, Greg deZarn-O'Hare, and all the people who turn a manuscript into a book. My thanks also to the sales and marketing staff who ensure that my efforts were not in vain. Last, I want to thank all of the O'Reilly authors whose company I am proud to be in.

This project would not have happened without the efforts of my agent, David Rogelberg of Studio B Productions (*http://www.studiob.com*). He was instrumental in the development and genesis of both *Director in a Nutshell* and *Lingo in a Nutshell*, for which I am forever grateful. My thanks also to Sherry Rogelberg and to the participants of Studio B's Computer Book Publishing list.

The quality of the manuscript reflects my excellent technical reviewers, all of whom made time for this semi-thankless job despite their busy schedules: Lisa Kushins, who verified items to an extent that astounded me and provided feedback that improved every chapter she touched; Hudson Ansley, whose keen eye and unique perspective also improved the book immeasurably; Mark Castle (*http://www.the-castle.com*), who helped shape the style and content from the earliest stages; and Matthew Pirrone and James Terry (*http://www.kandu.com*), who both provided excellent feedback on Chapter 4, *Lingo Internals*, and Chapter 20, *Lingo for C Programmers*, (downloadable from the web site). My thanks also goes out to all my beta-readers who provided useful feedback, most notably Miles Lightwood and Birnou Sdarte.

I cannot begin to thank all the Macromedians who develop and support Director, many of whom provide technical support on their own time on various mailing lists. My special thanks goes to Buzz Kettles for all his feedback regarding Shockwave audio and to Michael Seery for being my inside connection at Macromedia all these years. My thanks also to Lalit Balchandani, David Calaprice, Jim Corbett, Landon Cox, Ken Day, Peter DeCrescenzo, David Dennick, John Dowdell, Mike Edmunds, John Embow, Eliot Greenfield, Jim Inscore, David Jennings, James Khazar, Leona Lapez, S Page, Bill Schulze, Karen Silvey, Joe Sparks, John Thompson, Karen Tucker, Anders Wallgren (expatriot), John Ware, Eric Wittman, Doug Wyrick, and Greg Yachuk, all of whom fight the good fight on a daily basis.

My thanks go out to the wider Director community, including but not limited to Stephen Hsu, Brian "Bam Bam" Johansen, Peter Fierlinger, Brian Gray, Roger Jones, Tab Julius, Irv Kalb, Kathy Kozel, Alan Levine, Gretchen Macdowall, Myron Mandell, Kevin McFarland, Hai Ng, Roy Pardi, Darrel Plant, Peter Small, Kam Stewart, Stephen Taylor, Andrew White, John Williams, Alex Zavatone, all the participants of the super-secret mailing lists that I cannot name, and all the users who have given me feedback over the years, including the AOL old-timers.

Thank you also to Caroline Lovell-Malmberg, who can now forgive her husband, Mark, for leaving her out of his acceptance speech. Perhaps he'll thank whomever I've inadvertently left out next time he wins an Oscar.

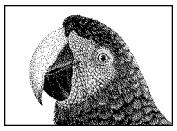
I'd like to thank you for taking the time to read this book. If I never get around to stand-up comedy, it is nice to know I still have an audience somewhere. If you enjoy the book, you owe a debt of gratitude to Professor David Thorburn, who taught me more about writing than anyone before or since.

Last, I want to acknowledge my entire family, whose sacrifices and support truly made this book possible. If this book saves you time that you can then devote to

your family, my efforts will not have been in vain. Good luck in all your multimedia pursuits.

Bruce A. Epstein
May 1998
Franklin Park, NJ

"All the love that you miss...all the people that you can recall...do they really exist at all?" — Lowell George, anticipating Virtual Reality by 20 years



CHAPTER 1

How Lingo Thinks

So You're Too Busy to Learn Lingo

Do you really have time to read a book on Lingo when you're facing a deadline? The answer depends on how much time you waste struggling with Lingo and how often you've compromised your Director projects for lack of Lingo skills. If you make the investment now, this book will pay marvelous dividends. It may save you weeks otherwise spent flailing over relatively trivial Lingo problems, and it can help you to add snazzy new features and a professional polish to your Director projects.

If you don't have a project to work on, pick one now. You will learn much more if you have a concrete goal and concrete problems to solve. You have been warned.

Learning to program is a process, not an event. Although this book is not a substitute for an introductory programming class, it covers basic, intermediate, and advanced topics. The material is very condensed, but the book also lavishes attention on topics that are omitted entirely from other Lingo books. Before proceeding, you should understand Director's Cast, Score, and media editing windows, as covered in Macromedia's *Using Director* manual. You might also want to skim Macromedia's *Learning Lingo* manual for a broad overview of Lingo.

Most books provide simple examples that leave you stranded when you try to accomplish your specific goals. This book teaches you how to do *anything* you want with Lingo, not just create simple clickable buttons. It provides a solid foundation instead of a house of cards, and it is for people who want to know more, not less. As such, this book explores many abstract concepts that may not be relevant to your immediate needs. You must exercise reasonable discretion by ignoring topics that don't interest you or are beyond your current level.

This chapter lays the groundwork for your Lingo-laden future, but the details of using Lingo to add interactivity are in later chapters (starting with Chapter 2, *Events, Messages, and Scripts*). You should first focus on understanding how Lingo itself "thinks." Lingo is a marathon, not a sprint, and the extra training will pay off in the long run. More practical examples are given in Chapter 9, *Mouse Events*, and Chapter 10, *Keyboard Events*. Refer to the companion book, *Director in a Nutshell*, for details on using Lingo to control and analyze cast members, sprites, sounds, digital video, MIAWs, fields, and memory.

You are not expected to understand the entirety of this book the first time you read it. Much of it will be meaningless until you've worked with Lingo for a few months and encountered specific problems that you wish to solve. At that time, you will recall enough to know what sections you need to reread. As in the film *The Karate Kid*, what may seem like meaningless manual labor is really your first step toward a black belt in Lingo. You should revisit this and other chapters periodically. They will reveal additional nuggets of knowledge as your experience and problems with Director and Lingo grow. Certainly, you should return to the appropriate chapter whenever you encounter a vexing problem, as the chances are high that the answer lies herein.

Even if Lingo is your first programming language, this chapter will help you to understand other people's Lingo code (which is the first step in creating your own). This chapter unavoidably introduces many new concepts that depend on other material not introduced until later (the old "chicken and the egg" problem). Skip around the chapter as necessary, and consult the Glossary whenever you feel queasy. Keep in mind that this chapter is intended to satisfy a broad range of users, some with much more programming experience than others. Skip the mind-numbing sections that don't have relevance for you yet (but revisit them later). Above all, do not lose heart. If you keep reading, you'll encounter the same concepts again, and they will eventually make sense. In the words of Owl, "Be brave, little Piglet. Chin up and all that sort of thing."

The example code used throughout this book is available from the download site cited in the Preface, but you should create a test Director movie file and type in the shorter examples by hand. Add the examples in each chapter to your test movie and use it like a lab notebook full of your experiments. This practice will make it much easier to write your own Lingo when the time comes. You might want to maintain a separate test movie for each chapter; start with a fresh movie (to eliminate potential conflicts) if an example doesn't seem to work.

You must abandon the safety of spoon-fed examples and experiment. If at first you don't *fail*, try, try again. You will learn more from failure than from success.

Experienced programmers can skim most of this chapter but should read the sections entitled "*Recursion*," "*Dynamic Script Creation*," "*The Classic Three-Line If Statement*," "*Special Treatment of the First Argument Passed*," and "*Variable-Length Parameter Lists*." Also see Chapter 4, *Lingo Internals*.

Let us set our goals high and see if we can stretch our minds to reach them. Let us now commit ourselves not only to learning Lingo, but also to becoming true *Linguists*, as fluent in Lingo as we are in our own native tongues.

Like all experienced Linguists, you should first build a shrine to the Lingo Gods with an altar for burning incense to summon and appease them. Abandon all hope, ye who enter here, for there is no turning back.

"Do or not do. There is no try."—Yoda

Lingo Scripting Basics

Computer languages tend to be simpler and more rigid than human languages, but like any other language Lingo has a set of rules that control the structure (*syntax*) of your Lingo program. Just as languages have grammar, Lingo's *syntactical* rules restrict the spelling, vocabulary, and punctuation so that Director can understand your instructions.

A *syntax error* or *script error* usually indicates a typographical error or the incorrect use of a Lingo statement.

Lingo's built-in *keywords* (or *reserved words*) make up Lingo's vocabulary and are the building blocks of any Lingo program. We'll see later how these keywords form the skeleton of your Director program, just as any language's words are the basis for sentences and paragraphs. It is crucial that you recognize which items in a Lingo script are built-in keywords versus those that are specified arbitrarily by the programmer. Refer to Chapter 18, *Lingo Keyword and Command Summary*, for a complete list of all Lingo keywords. The *PrettyScript* Xtra (*http://rampages.onramp.net/~joker/tools/*) is a $20 U.S. shareware tool that colorizes some items in your Lingo scripts to make them easier to recognize. The *ScriptOMatic Lite* Xtra, is available under `XtrasScriptOMatic➤Lite`, colorizes a broader range of items, but it is crippled almost to the point of being useless. The full version is promised imminently from g/matter (*http://www.gmatter.com/products/scriptomatic/*) at press time.

Handlers and Scripts

A *handler* is a series of Lingo statements that tell Director to perform some useful function. Handlers are typed into *script* cast members in the Script window. ("Script" is also used loosely to refer to a handler within a script. "Code" is used both as a noun to indicate your Lingo scripts and as a verb, meaning "to program" or "to write Lingo scripts.")

The scripts in a Director movie control the action, just as real-life actors follow a script. There are several types of scripts (castmember scripts, movie scripts, sprite scripts, parent scripts, and Behaviors), which are covered in detail in the "*Lingo Scripts and Handler Types*" section of Chapter 2.

Hello World

As required by the International Programmers' Treaty of 1969, we'll start with an example that displays "Hello World." Open up a *movie script* cast member using `Cmd-Shift-U` (Macintosh) or `Ctrl-Shift-U` (Windows).

Enter Example 1-1 exactly as shown into the movie script window.

Example 1-1: Hello World

```
on helloWorld
  alert "Hello World"
end
```

The keyword *on* identifies the beginning of our handler, which we *arbitrarily* chose to name *helloWorld*. The keyword *end* signifies the end of our handler.

The examples beginning with the word *on* are handlers that must be typed into a script, not the Message window.

With minor exceptions, your Lingo code for each handler goes between the *on* `handlerName` and *end* commands (see "*Where Commands Go*").

Handler names must be one word, but they are case-insensitive, so you can use capitalization to make them easier to read. Name your handlers descriptively so that you can remember what they do, and as a rule you should avoid naming them the same as existing Lingo commands (see Table 18-1).

A handler name must start with an alphanumeric character, not a digit, but it can contain digits, decimal points, and underscores. Only the first 260 characters of the name are significant.

Movie script cast members are simply repositories for our handlers and are not used in the Score (see Chapter 2 for details on score scripts).

Entering a handler into a script (as shown above) *defines* or *declares* the handler and is referred to as a *handler definition*. Defining (declaring) a handler makes it available for future use, but the handler doesn't execute until something tells Director to run it for you.

Close the Script window to *compile* it (that is, to prepare it to run). When the handler is run (*called*), Lingo will execute each line (that is, each *command*) in the order in which it appears in the handler. There is only one command in our *helloWorld* handler; the built-in *alert* command displays the specified text in an alert dialog box.

The *Message window* provides an area for printing messages from Lingo and testing Lingo scripts (see Chapter 3, *Lingo Coding and Debugging Tips*). A handler stored in a movie script can be executed (called) by typing its name in the Message window (or by using its name in another handler).

Open the Message window using `Cmd-M` (Macintosh) or `Ctrl-M` (Windows). In the Message window, type the name of the handler to test (`helloWorld` without any

spaces). Do not precede the name with the word *on*, which is used only to declare a handler, not to run it.

```
helloWorld
```

Always press the `RETURN` key (Macintosh) or the `ENTER` key (Windows) at the end of the line to initiate the command. Example code shown flush left should be typed into the Message window, as opposed to handler definitions that are entered in the Script window.

Congratulations, you are now a Lingo programmer! After accepting your diploma, please step to the right. If your script didn't work, make sure you typed everything correctly and that you entered the script in a *movie* Script window (not a score script, castmember script, field, or text cast member). Choose `Control➤Recompile All Scripts`. If it still fails, hang your head in shame, or see Chapters 3 and 4.

Calling All Scripts

Typing `helloWorld` in the Message window *calls* (locates and runs) the handler of the same name. Reopen the script, and change both the name of the handler in the script and the name you type in the Message window to something new. If the names don't match, what happens? Did you remember to recompile the script by closing its window? Set the *Using Message Window Recompiles Scripts* option under `Preferences➤General` to ensure that the latest version of a handler is executed. See "*Compiling Scripts*" later in this chapter.

Note above that `"Hello World"` is automatically incorporated by the *alert* command into the alert dialog. You can change the displayed text by specifying any *string* (series of characters) in place of `"Hello World"` (don't forget the quotes). The specified string is said to be an *argument* to the *alert* command, and it is used to customize the dialog. See "*Commands and Functions*" and "*Parameters and Arguments*" later in this chapter for complete details on using arguments with built-in Lingo commands and custom handlers.

Previously we created an arbitrarily named custom handler and called it from the Message window by using its name.

You can add more handlers after the end of the *helloWorld* handler in the movie script used above, or you can press the "+" button in the Script window to create a second movie script. (You can have a virtually unlimited number of movie scripts).

Naturally, the user will not be typing anything in the Message window. When the user clicks the mouse button or presses a key Director tries to run handlers named *on mouseDown*, *on mouseUp*, *on keyDown*, and so on. In practice, you'll create *event handlers* with these reserved names to respond to user events. If you name

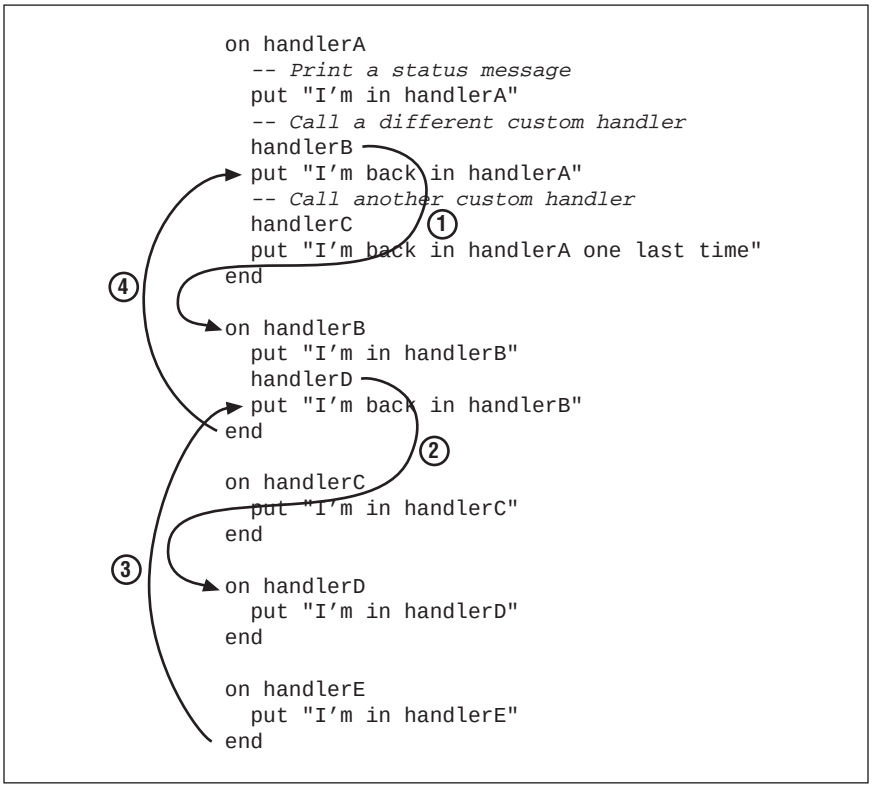
the handlers incorrectly, they will never be executed. See the "*Events*" section in Chapter 2, and Chapter 9 for more details.

Nested Handler Calls

Just as we can call a handler from the Message window, one handler can call another simply by using its name. As each handler completes its work, control returns to the calling handler. You can picture a hierarchy of *nested* handler calls as an outline that periodically is indented further, then returns to the previous level. Suppose we define several handlers (some of which call other handlers) in a movie script as shown in Example 1-2.

The *put* command prints a message in the Message window and is used throughout the book to peek inside Lingo. The *&&* and *&* operators are used to assemble long strings (see Chapter 7, *Strings*). Lingo lines starting with two hyphens ("--") are comments for the reader's benefit (see "*Comments*," later in this chapter).

Example 1-2: Nested Handler Calls



We can then test it from the Message window.

```
handlerA
-- "I'm in handlerA"
-- "I'm in handlerB"
-- "I'm in handlerD"
-- "I'm back in handlerB"
-- "I'm back in handlerA"
-- "I'm in handlerC"
-- "I'm back in handlerA one last time"
```

The series of Lingo handlers that are currently "pending" is known as the *call stack*. The call stack is always shown in the upper left pane of the Debugger window (see Figure 3-1).

Note that *handlerA* calls *handlerB*, which then calls *handlerD*. Control then passes back through *handlerB* and back to *handlerA*. Finally, *handlerA* calls *handlerC*, and control is then returned to *handlerA*. Conceptually, this can be pictured as:

```
handlerA
  handlerB
    handlerD
  handlerC
```

Note that the order of execution *within* a handler is determined by order of the Lingo commands, but the order in which the four handlers are typed into the movie script is irrelevant. Execution begins at our so-called *entry point* (in this case *handlerA*, which was called from the Message window), and Lingo takes detours into each called handler before it returns control to the calling handler. Note that *handlerE* is never called and therefore never executed, even though it has been defined in the same script with our other handlers.

Recursion

Each time a handler is called, a *copy* of it is created temporarily. The copy comes into existence when the handler is called and disappears when the handler completes. You can call a handler hundreds of times, and each occurrence will be independent of the other occurrences. A handler that calls itself, as shown in Example 1-3, is called a *recursive function*, and this technique is called *recursion*.

If we call our recursive function from the Message window, Director will run out of memory. (Save your work before testing this as it may crash your machine.)

Example 1-3: A Recursive Function

```
on recurseTest
  recurseTest
```

Example 1-3: A Recursive Function (continued)

```
  put "End of recurseTest reached"
end recurseTest
```

If you are using recursion, it is probably an accident. As a general rule, you should avoid it. It is like a reflection repeated infinitely between two mirrors; in the extreme case it will crash Director. See Example 6-9, "*Recursively Sorting Sublists* in Chapter 6, *Lists*, Example 8-16, "*Recursive Factorial Function*" in Chapter 8, *Math (and Gambling)*, and Example 14-5, "*Extracting Files in Subfolders*" in Chapter 14, *External Files*, for proper uses of recursion.

Even if we call *recurseTest* only once, it calls itself repeatedly so that Director keeps going "down" an extra level and never comes back up for air. Director will run out of memory before the *put "End of recurseTest reached"* command is ever reached. Note that each time *recurseTest* is called, it spawns another copy of itself. Conceptually, this can be pictured as follows:

```
recurseTest
  recurseTest
    recurseTest
      recurseTest
        (ad infinitum until Director runs out of memory)
```

Note that it is perfectly acceptable for one handler to call another repeatedly, as is often done using a *repeat* loop (see "*Repeat Loops*" later in this chapter).

Example 1-4: Calling a Function Non-recursively

```
on testIt
  repeat with i = 1 to 100
    someHandler
  end repeat
end

on someHandler
  put "I am inside someHandler"
end
```

Typing `testIt` in the Message window will print out "*I am inside someHandler*" 100 times with no ill effects because each time *someHandler* completes, control is returned to the top level (in this case, the *testIt* handler).

Entering and Compiling Scripts

There is no magic to entering Lingo scripts. Scripts are typed in script cast members (or attached to non-script cast members) via the Script window. Script cast members appear in the Cast window along with your other assets (bitmaps, fields, and so on). The Script and Message windows include buttons to access pop-up menus of Lingo commands (both alphabetical and by category). You can use these to insert commands into your scripts or to remind you of the correct syntax. Refer to Chapter 2 of this book and to Chapter 2, *Script Basics*, of Macromedia's *Learning Lingo* manual for details on creating scripts and entering your Lingo.

Where Commands Go

All your Lingo code goes between a handler's *on* `handlerName` and *end* statements.

The exceptions to the rule are:

- Each handler should be separate from other handlers. Handler declarations are not "nested" the way that *if...then* statements can be. Do not start a new handler before *end*'ing the first one, as described under "*Common Handler Declaration Errors*" later in this chapter.
- Comments can be placed both inside and outside handlers.
- Property variables must be declared outside any handler.
- Global variables can be declared both inside and outside handlers. Global variables declared within a handler apply only to that handler. Global variables declared outside a handler apply to all handlers in the script following the declaration.

Each Lingo command occupies its own line (although there are some multiline Lingo statements, discussed under "*Multiline Code Structures*" later in this chapter). Each line of Lingo is separated using a carriage return—that is, using the `Return` key (Macintosh) or `Enter` key (Windows) on the main keyboard, not the one on the keypad.

You can enter long lines of Lingo code in your script without line breaks; Director will wrap them automatically. To improve readability (as with many examples in this book), long lines of Lingo code can be continued onto the next line using the Lingo continuation character ¬, as shown in the example that follows. This special character is created using `Option-Return` (Macintosh) or `Alt-Enter` (Windows).

```
-- Here is a long line of Lingo broken onto two lines
set the member of sprite (the currentSpriteNum) = ¬
   member "Hilighted Button"
```

You can break long lines of Lingo onto more than two lines using an ¬ character at the end of each line (except the last) of the long command. Do not break the long lines (that is, do not use the ¬ character) within quoted strings (see Chapter 7, *Strings*). Do not put anything, even a space, on the same line after a continuation character.

Director ignores leading spaces and automatically indents your Lingo code according to its own rules. For example, all lines within a handler between the *on* and *end* statements are indented at least two spaces. Use the same font and type size throughout the script to make the code more legible and indentation problems obvious. Colorized or formatted text can slow the Script window's response, especially for longer scripts. A single script cast member is limited to 32,000 characters, but you can use as many script cast members as required.

Use the `Tab key` to automatically indent your Lingo code. If the indentation is wrong, you may have omitted necessary keywords or used them in the wrong order. Refer to "*Lingo's Skeletal Structure*" later in this chapter.

The Lingo code you enter is simply text (although you should enter it in a script cast member, not in a text or field cast member). Before it can be run, Director must *compile* your Lingo code into a machine-readable format. (This is analogous to a player piano, which cannot read sheet music but can play a song if is transcribed onto a paper roll.)

When Director compiles a script, it checks that the syntax conforms to the accepted rules and does its best to *parse* (understand) the structure of your Lingo. Compilation is analogous to spellchecking and grammar checking in a word processor. It merely checks that your Lingo code has a recognizable structure and acceptable keywords. It does *not* attempt to actually run your Lingo code.

It would be counter-productive for Director to attempt to compile your scripts as you type them. Use `Control➤Recompile All Scripts` to compile your scripts when you finish typing (see "*Compiling Scripts*" in Chapter 2).

If Director's compiler fails, it displays a script error (a syntax error) that identifies the offending portion of the Lingo, but it may merely reflect a problem that lies elsewhere. You would then correct the Lingo and recompile. If successful, it creates a hidden compiled version of your Lingo script that runs more quickly than it would if Director had to reinterpret your human-readable script every time it runs.

If compilation succeeds, your code is not necessarily error-free and may still cause a so-called *runtime error* when Director attempts to run it. (In this context *runtime* refers to when the script is *executed*, as opposed to when it is *compiled*. This should not be confused with *authoring time* (in Director) vs. *runtime* (in a Projector). Refer to Chapter 3 for more details.

Handler Scope

Handlers are stored in script cast members (excepting those attached directly to other member types); the multiple types of script cast members are explained in great detail in Chapter 2. The script cast member's type (*movie script*, *score script*, or *parent script*) affects the *scope* of all the handlers declared within it (that is, which other scripts can "see" these handlers and from where they are accessible). We were able to test the example handlers above from the Message window because they were entered in *movie* scripts. (A *movie script* is a script cast member whose type is set to *Movie* in the script cast member's info window). Handlers in movie scripts can be "seen" from the Message window or any other script in the same Director movie (even from scripts in other linked castLibs) because they have *universal* scope.

Had we entered the example handlers in *score* scripts, attempting to use them from the Message window would result in a "*Handler not defined*" error because the scope of score scripts is more limited.

If two handlers in the *same* script cast member have the same name, Director will complain. Neither should you use two handlers with the same name in two *different* movie scripts because the first handler found will intercept all function calls using that name, and the second handler will always be ignored.

Place any handlers that you use in multiple Director movies or multiple Director projects into movie scripts in an external cast library that you can link into your project. Use unique names, perhaps starting with a prefix such as "*lib*," that are unlikely to conflict with other handlers in a given movie.

Avoid naming your handlers the same as existing Lingo commands (see Table 18-1). A custom handler (stored in a movie script) that has the same name as a built-in Lingo command will intercept (and override) any calls to that Lingo command. If accidental, such an error can be extraordinarily hard to debug.

Contrary to movie scripts, it is very common to use handlers of the same name in *score scripts*. (Again, these are explained in detail in Chapter 2.) The important point is that the handlers in score scripts have a different scope (accessibility) than handlers in movie scripts. For example, most sprite scripts (one type of score script) will contain *on mouseUp* handlers, and most frame scripts (another type of score script) will contain *on exitFrame* handlers. The same handler name can be used in multiple score scripts because they do not have universal scope as do handlers in movie scripts. Director automatically calls only those handlers that are attached to the current frame or the sprite on which the user clicked. Other *on mouseUp* and *on exitFrame* handlers located in other score scripts won't interfere. Refer to Chapter 9 for more details. Likewise, Lingo calls an *on keyDown* handler only if it is attached to the field sprite that has keyboard focus (see Chapter 10).

Example 1-5 demonstrates the different scope of a handler depending on the script type in which it exists.

If the following two handlers coexist in the same *score* script cast member, *handlerA* can call *handlerB* (or vice-versa).

Example 1-5: Handler Scope

```
on handlerA
  handlerB
end

on handlerB
  put "I am doing something, so please stop hovering."
end
```

If the two handlers existed in two separate *score* scripts, however, they would not "see" each other and therefore could not call each other. On the other hand, if *handlerA* was in a *score* script, but *handlerB* was in a *movie* script, *handlerA* could call *handlerB*, but *not* vice-versa. Furthermore, if *handlerB* is in a movie script, it can be called from other handlers in other scripts of any type. Therefore, you should place one copy of your general-purpose utility handlers in a movie script rather than replicating it in multiple score scripts.

Handlers in movie scripts can be called from anywhere at any time and are usually custom handlers named arbitrarily by the programmer. Handlers in score scripts are generally named to respond to predefined events (such as *mouseUp*) and are called by Director in response to those events.

This example offers a simplified picture of the universe. In actuality, any handler in any script can be called from anywhere if you refer to the script explicitly. You usually refer only to the handler name and let Director decide in which script to find the handler. This is covered in Chapter 2, along with details on the way that handlers in multiple scripts are sometimes called in succession.

See "*Special Treatment of the First Argument Passed*" later in this chapter for details on how the first argument to a function affects which scripts are searched for a matching handler.

Commands and Functions

A *command* tells Director to do something, such as play a sound, but usually does not return any result. Built-in Lingo keywords are referred to as *commands*, but you can create custom handlers that are used just like the built-in commands, essentially extending Director's command set. (The word *command* is also used loosely in many contexts, including to indicate a menu choice.)

The general format of a command is:

```
commandName arg1, arg2, arg3, ...
```

where the arguments (*arg1*, *arg2*, *arg3*, ...) are inputs used by the command, and may be optional or mandatory, and vary in number and type depending on the command. For example, the *alert* command shown previously expected a single string argument. The *puppetSprite* command expects two arguments (an integer and a Boolean value), as in:

```
puppetSprite 17, TRUE
```

A command that returns a result is called a *function* (the terms, though, are often used interchangeably). The result may be a number, a string, or any other data type. The general format of a function is

```
set variableName = functionName (arg1, arg2, arg3, ...)
```

or

```
put functionName (arg1, arg2, arg3, ...) into variableName
```

where again the arguments (*arg1*, *arg2*, *arg3*, ...) may be optional or mandatory and may vary in number and type depending on the function.

For example, the *power()* function requires two arguments and raises the first argument to the power specified by the second argument. You wouldn't ordinarily compute a value and discard the result; you would either print it out in the Message window or store it in a *variable* (a container for data). Below, the result of the calculation is stored in a variable that is arbitrarily named *myValue* (see "*Variable Storage Classes and Data Types*" later in this chapter for details on variables).

```
set myValue = power (10, 2)
```

If you don't store the result in a variable, the function still returns a result that can be used in other expressions (see Chapter 5, *Data Types and Expressions*). This prints the result in the Message window instead of storing it in a variable:

```
put power (10, 2)
-- 100.0000
```

This uses the result of the calculation to decide whether to print a message:

```
if power (10, 2) > 50 then put "That's a big number."
```

In some cases, Director issues a "*Command not defined*" error if you use a function by itself rather than as part of a larger expression:

```
power (10, 2)   -- This causes an error
```

Either use *put power (10, 2)* to print the result of the function call in the Message window or assign the result to a variable, as shown earlier.

If a function does not require any arguments, you must still include the parentheses to obtain the result, such as:

```
put browserName()
-- "Mac HD:Netscape Navigator Folder:Netscape Navigator"
```

See "*Return Values and Exiting Handlers*" later in this chapter for details on returning values from custom handlers.

Lingo allows *nested* function calls, in which the result of one function is used as an argument to another function, such as:

```
if (random(max(x, y)) > 5) then ...
```

In such a case, the result of *max(x, y)* is used as an argument to the *random()* function. The preceding code is just shorthand notation for:

```
set myValue = max(x, y)
if (random(myValue) > 5) then ...
```

Return Values and Exiting Handlers

This section is next in the logical progression of the chapter, but it will not make sense unless you understand concepts explained later. You can skim it now and revisit it after reading the rest of the chapter. As alluded to earlier, a handler often

performs a calculation and returns the result to the calling routine. A handler or Lingo command that returns a value is called a *function*. Most functions require some inputs on which to operate (see "*Parameters and Arguments*" for details). For example, the built-in Lingo function *max()* returns the maximum value from the list of items you send it:

```
put max (6, 9, 12)
-- 12
```

You might write your own custom function that returns TRUE or FALSE based on whether the input is a valid digit between 0 and 9, as shown in Example 1-6.

Example 1-6: Returning a Value from a Function

```
on isDigit someChar
  if "0123456789" contains string (someChar) then
    return TRUE
  else
    return FALSE
  end if
end isDigit
```

In this case, the result (1) signifies the Boolean value `TRUE`:

```
put isDigit (5)
-- 1
```

The parentheses surrounding the arguments are mandatory when calling a function that returns a value. Even if the function does not require any parameters, you must still include the parentheses to obtain a result.

If the parentheses are omitted, Lingo would treat `isDigit` as if it were a variable name (see "*Variables and Properties*" later in this chapter) rather than a function name. In the following case, `isDigit` is mistaken for a `VOID` (valueless) variable, and the *put* statement prints the values `VOID` and 5 instead of the desired result of the function call.

```
put isDigit 5
VOID 5
```

Note the use of parentheses following rollover():

```
put rollOver()     -- rollOver() is treated as a function call
-- 7
put rollOver       -- rollOver is treated as a variable name
-- VOID
```

Leaving the Current Handler

Ordinarily a handler terminates when the last statement in it is reached (the *end* statement). Control then returns to whatever called the handler (either another handler or Director itself). In that case, no value is returned to the calling handler,

and any calculated result would be lost unless it were stored in a global or property variable. The *return* and *result* commands obtain return values from a handler. The *abort* and *exit* commands terminate a handler prematurely. (They differ from *next repeat* and *exit repeat*, which affect program flow but do not exit the handler). The *quit*, *halt*, *restart*, *shutDown*, *go*, *play*, *play done*, and *pass* commands may also affect the order of Lingo execution.

`return`

The *return* command exits the current handler and returns control to the calling routine, along with an optional value of any data type. Any statements following the *return* are not executed, which makes *return* convenient for exiting a handler once a particular condition is met or task is completed.

Example 1-7 returns as soon as it finds a sound cast member in the primary castLib. It returns zero (0) if no sound cast member is found. The other details are not important at this point.

Example 1-7: Returning Prematurely from a Function

```
on findFirstSound
   -- Loop through the cast
  repeat with n = 1 to the number of members
    -- Look for a sound castmember
    if the type of member n = #sound then
      -- Return the number of the sound
      -- and we're out of here!
      return n
    end if
  end repeat

   -- If no sound  was found, return 0
  return 0
end findFirstSound
```

Test it in the Message window:

```
put findFirstSound()
-- 72
```

The above technique is best used with small handlers. Avoid using multiple *return* statements to exit a large handler from many different points. It makes the code harder to understand and maintain. Storing the eventual return value in a variable and returning it at the end of the handler is often clearer.

You can use *return* without a value, in which case it is identical to *exit*, and the caller receives `VOID` as the returned value. Note that the *return* command is distinguished by context from the `RETURN` constant (which indicates the carriage return character).

`the result`

The result retrieves the result of the last function call, even if it was never stored in a variable when it was returned by the last function call.

```
set x = isDigit (5)
put x
-- 1
isDigit (5)
put the result
-- 1
```

Some commands, such as *preLoad* and *preLoadMember*, do not return a value, but set *the result*.

```
preLoadMember 1, 5
put the result
-- 5
```

`abort`

Abort aborts the *call stack* (that is, the current Lingo handler *and any handlers that called it*) without executing any of the remaining Lingo statements in any of those handlers. By contrast, *exit* exits only the *current* handler. *Abort* is useful when you want to stop the current action in response to some drastic change. For example, if the user presses the `Escape` key, you may abort what you are doing:

```
on getUserInfo
  -- Abort if the user hits ESCAPE (ASCII code 27)
  if the key = numToChar (27) then
    abort
  end if
end getUserInfo
```

Abort does not quit Director (see *halt* or *quit*), nor does it abort asynchronous operations in progress (see *cancelIdleLoad*, *netAbort*). *Abort* aborts only Lingo execution; it does not affect the Score's playback head (see the *pause* command).

`exit`

Exit (not to be confused with *exit repeat*) causes Lingo to exit the current handler. It exits "up" only one level to the calling handler, as opposed to aborting the entire call stack (see "*Nested Function Calls*" earlier in this chapter). Exit is often used to exit the current handler if some condition is not met:

```
on playVideo
  if not (the quickTimePresent) then
    alert "You can't play video without QuickTime."
    exit
  end if
  -- Remaining statements are run only
  -- if QuickTime is installed.
end playVideo
```

When using *exit*, no return value is sent. Use *return* instead to return a value to the calling handler.

`quit`, `halt`, `restart`, *and* `shutDown`

Quit and *halt* immediately quit a Projector and are generally used only in a script attached to a Quit button or at the end of a presentation. During development, *halt* stops the movie without quitting Director. See Chapter 8, *Projectors and the Run-time Environment,* in *Director in a Nutshell* for details on these and other commands, including *restart* and *shutDown*.

`go`

The playback head moves semi-independently remaining of Lingo commands. If you use the *go* command to move the playback head, commands in the handler are still executed *before* jumping to the new frame.

```
on exitFrame
  go frame 50
  put "This will be printed before jumping to frame 50"
end
```

`play` *and* `play done`

The *play* command works differently than the *go* command. Commands following the *play* command are executed, but not until a *play done* command returns control back to the original script. Commands following *play done* are never reached. Test this using a frame script of the form:

```
on exitFrame
  play frame 50
  put "This will be printed second, not first"
end
```

In frame 50, use this frame script to complete the test:

```
on exitFrame
  put "This will be printed first"
  play done
  put "This line won't ever be reached or printed"
end
```

`pass`

The *pass* command aborts the call stack (see the *abort* command). Commands following the *pass* command are never executed. Control immediately jumps to the next script that handles the event being passed (see Chapter 2). For example:

```
on mouseUp
  pass
  put "This line won't ever be reached or printed"
end
```

Dynamic Script Creation

This section is next in the logical progression of the chapter but is fairly advanced—and irrelevant for most users. You can skim it or even ignore it altogether without significant regret. You can create new Lingo dynamically at runtime (that is, while Director or even a Projector is running). Using the *do* command or by setting *the scriptText of member* property, you can actually create new Lingo from within Lingo! You can dynamically evaluate a string as if it is a Lingo statement using the *value()* function. Most programming languages don't allow this,

and you will rarely use this feature. The following examples are for illustration only and do not necessarily depict likely uses.

Do Statements

The *do* command can compile and execute a string on the fly as if it were a Lingo statement. Although it should not be used haphazardly, it can perform some interesting tricks. Most notably, you can use it to execute an arbitrary command stored in a text file or a field cast member. You can create a pseudo-Message window for debugging Projectors by *do*'ing the text entered in a field cast member:

```
do "beep"
do the text of field "someFieldCastMember"
```

You cannot use *do* to declare global variables without a trick. The following will not work:

```
do "global gSomeGlobal"
do "set gSomeGlobal = 5"
```

To declare a global with a *do* statement, use:

```
do "global gSomeGlobal" & RETURN & "set gSomeGlobal = 5"
```

To make use of a global within a *do* statement, declare the global inside the current handler. So-called "global globals" declared outside the currrent handler are not recognized during a *do* statement. The following example is illustrative only and would never be required in reality:

```
on setGlobal
  global gSomeGlobal
  do "set gSomeGlobal = 7"
end
```

Value Statements

The *value()* function can be used to convert a string into an integer or a float, as is useful when converting string data from a file or user input into a number.

```
set userEntry = value (field "Age in Years")
```

Value() can also evaluate a string or symbol as if it is a variable name:

```
set someVar = "Oh happy days!"
put value ("someVar")
-- "Oh happy days!"
put value (string(#someVar))
-- "Oh happy days!"
```

Setting the ScriptText at Runtime

The *scriptText of member* property contains the Lingo code from a script cast member. The human-readable form of *the scriptText* is stripped when creating Projectors and protecting DIR files (creating DXR files), leaving only the hidden

internal compiled version. Even from a Projector, you can create a script dynamically, as shown in Example 1-8.

Example 1-8: Creating Scripts at Runtime

```
-- Set up a string variable containing the text:
on createscript
  set dynaScript = "on newHandler" & RETURN ¬
                   "global gMyGlobal" & RETURN ¬
                   "set gMyGlobal to 52" & RETURN & "end"
-- Create a new movie script cast member, and fill it in.
  set newMovieScriptCastMember = new(#script)
  set the scriptType of newMovieScriptCastMember = #movie
  set the scriptText of newMovieScriptCastMember = dynaScript
end
-- Now you can run it
createscript
newHandler
-- If desired, you can then delete the cast member
erase newMovieScriptCastMember
```

Variables and Properties

You'll often read Lingo properties to obtain information about the user or the run-time environment and set properties to affect the run-time environment. You'll use variables as containers to store and manipulate any type of data, including Lingo properties. Later we'll see how you can define your own properties using so-called *property variables*, which are a particular type of variable but are unrelated to Director's built-in properties.

Built-In Lingo Properties and Constants

Lingo defines dozens of *properties* (not to be confused with programmer-defined property variables, discussed later), which are always preceded by the keyword *the*.

If you omit the word *the*, Lingo thinks you are referring to a programmer-defined variable, not a Lingo property.

A property may pertain to the overall system, the current movie or MIAW, a cast member, or a sprite. Lingo properties are universally accessible (they can be used from any script at any time), and they may be *static* (fixed) or may change over time or based on user actions. For example, *the platform* property doesn't change unless you switch computers, but *the mouseH* and *the mouseV* properties change whenever the cursor moves.

The general format for setting a Lingo property is:

```
set the property {of object} = value
```

or

```
put value into the property {of object}
```

where *value* is replaced with a meaningful value, *property* is a property name and *object* is usually a sprite, member, or window reference. The following examples set *the locH of sprite* sprite property, *the regPoint of member* member property, and *the soundEnabled* system property. Note that all three properties start with *the*, although no *object* is specified for *the soundEnabled* property because it is a system property, not a property of a sprite or a cast member.

```
set the locH of sprite 1 = 17
set the regPoint of member 1 = point(0,0)
set the soundEnabled = TRUE
```

The general format for reading a Lingo property is:

```
set variableName = the property {of object}
```

such as:

```
set spritePosition = the locH of sprite 1
set memberDepth = the depth of member 1
set mousePosition = the mouseH
```

Some Lingo keywords are *constants* (fixed values) that don't use the word *the*, such as PI, TAB, and SPACE. See Table 5-6.

Common Errors When Using Properties and Constants

The following are common errors when using properties. Refer also to "*Common Errors When Using Variables*" later in this chapter.

Forgetting the word the:

The following sets stage white by setting *the stageColor* property to zero:

```
set the stageColor = 0
```

However the following defines a local variable, *stageColor*, and assigns it the value zero, which is probably not the desired goal.

```
set stageColor = 0
```

Using a property without the preceding *the* will cause a "*Variable used before assigned a value*" error, such as:

```
if mouseV > 50 then put "The mouseV is greater than 50"
```

(This won't fail in the Message window, but it will fail from within a script).

Confusing a local variable with a property of the same name:

Here, *mouseH* is a local variable, whereas *the mouseH* is a Lingo property. As the mouse continues to move, the property *the mouseH* will change, but the variable *mouseH* won't change without your explicit instruction.

```
set mouseH = the mouseH
```

Use variable names that are different from Lingo property names to avoid confusion, such as:

```
set lastMouseH = the mouseH
```

Confusing a variable with a Lingo constant of the same name:

Some keywords are reserved *constants* (see Table 5-6) that never change in value, even if you inadvertently try to assign a new value to them. For example, you cannot change the value of *pi*.

```
set pi = 54.36
put pi
-- 3.1416
```

Some properties can not be set:

Many properties can be both set and read, but some can be only read. The "*Cannot set this property*" error may result if you attempt to set a property that is read-only, such as *the mouseH*. Other properties may appear to be settable, but setting them may have no effect. For example, setting *the colorDepth* under Windows, or to an invalid value on the Macintosh, will leave the monitor depth unchanged, although no error results.

Using a "stale" value of a property that has changed:

Many Lingo properties change based on conditions beyond the programmer's control. For example, *the shiftDown* property changes whenever the user presses or releases the `Shift` key, and it may even change during the execution of, say, your *on keyDown* handler. If necessary, store the current value of a property in a variable (see details that follow), such as:

```
set shiftKeyState = the shiftDown
```

Version

There is a single Lingo global variable (global variables are explained later) named *version* that returns Director's version number (as a string) and can not be set nor cleared with *clearGlobals*. Do not use the name "version" as a variable name.

Use *version* as follows:

```
on testVersion
  global version
  put "The current version is" && version
end testVersion

testVersion
-- "The current version is 6.5"
```

Director also supports the property *the productVersion*, which doesn't require a *global* declaration (although the two methods return different values in Shockwave).

```
put the productVersion
-- "6.5"
```

Using Variables

A *variable* is a *container* for any type of data (such as an integer or a string) that usually stores values that may change or are not known until runtime. You are the master of your own variables. You can create as many as you need and give them

whatever names you like. For example, you might store the user's name and his high score in separate variables named *userName* and *highScore*.

Variables are *not* the algebraic "unknowns" that gave you nightmares in school. They are just convenient placeholders that allow your scripts to be flexible. Variables, unlike built-in Lingo properties, change only at your behest.

Once you've stored a value in a variable (see the next section) you can obtain that value simply by referring to the variable by name. See "*Data Types and Variable Types*," "*Type Assignment and Conversion*," and "*Constants and Symbols*" in Chapter 5 for more details on variables.

Assigning a Value to a Variable, Property, or Field

You store data in a variable, property, or field by *assigning* a value to it in one of the following ways:

```
set item = value
set item to value
put value into item
```

where *item* is a variable name, property (such as *the colorDepth*), or a field reference (such as *field "myField"*).

I *strongly* recommend the first form (*set item = value*) because it clearly delineates the variable or property on the left side of the expression from the value being assigned to it on the right side of the expression. For example:

```
set x = 5
set x = char 1 to 5 of "hello there"
set the soundEnabled = TRUE
set the loc of sprite 5 = point (50, 200)
```

I mention the other forms so that you will understand examples that make use of them, although I find these equivalent expressions harder to decipher:

```
put 5 into x
set x to char 1 to 5 of "hello there"
put TRUE into the soundEnabled
set the loc of sprite 5 to point (50, 200)
```

You *must* use the *put...into* form when replacing part of a string:

```
set x = "helloWorld"
put "H" into char 1 of x
```

These *won't* work:

```
set char 1 of x = "H"
set field 4 = "Some String"
```

But these *will* work:

```
set the text of field 4 = "Some String"
```

```
put "Some String" into field 4
```


Common Misconceptions About Variables

Some Lingo keywords used to assign variables are also used in other ways. Don't confuse *put...into* (which sets a value) with *put* by itself (which prints the value in the Message window).

```
put 5 into x --  assigns the value 5 to the variable x
put x        --  prints the value of x in the message window
-- 5
```

The keyword *to* is also used in chunk expressions, such as *char 1 to 5 of someString*. In the example below, the first *to* is used to perform the assignment, but the second *to* is used to indicate a range of characters.

```
set x to char 1 to 5 of "hello there"
```

The equals sign (=) is used for both *assignment* and *comparison*. Don't confuse the two uses. In C, the single equals sign is used only for assignment, and the *double* equals sign (==) is used for comparison (see the online Chapter 20, *Lingo for C Programmers*, downloadable from *http://www.zeusprod.com/nutshell/chapters/lingoforc.html*).

```
set x = 5  --  assigns the value 5 to the variable X
if (x = 5) then put "Yeah"  -- Compares x to the value 5
```

The equals sign assigns a value to an item; it does *not* indicate an algebraic equality. In algebra, the following would be meaningless because something can never equal one more than itself:

```
x = x + 1
```

On the other hand, the following is perfectly legitimate and is used frequently in programming.

```
set x = x + 1
```

How is this possible? The right side of the expression always uses the current (old) value for an item. The left side sets the *new* value for an item. The above example means "Take the current value of the variable ***x*** and add one, and then store the result back into ***x*** again." So:

```
set x = 4
set x = x + 1
put x
-- 5
```

An assignment statement is not an algebraic equation to be solved. Only one item can appear on the left side of the following assignment statement:

```
set x = 5 - y
```

This is not valid:

```
set x + y = 5
```

Although the two statements above may appear algebraically equivalent, they are not programmatically equivalent. The first one says "Subtract the value of ***y*** from 5,

and then store the result into the variable `x`." The second one, however, is trying to say "Set `x` and `y` so that they add up to 5." This confuses a computer because it wouldn't know whether to set `x` to 4 and `y` to 1, or `x` to 3 and `y` to 2, or one of the infinite number of alternatives.

Variable Types

A variable's *storage class* (local, parameter, global, or property) determines its initial value, its *scope* (by whom it can be accessed), and whether it *persists* (retains its value) over time. Don't confuse a variable's storage class, often called its *type*, with the data type of its *contents*. A variable's *data type* (integer, string, list, etc.) depends solely on the value assigned to it and is independent of its storage class. See Chapter 5.

There are four main storage classes (although *parameters* are generally treated as *local* variables), each of which is created in a different way.

Local Variables

Local variables (or *temporary variables*) are fleeting; they come into existence when they are first assigned a value, and they disappear at the end of the current handler. Use local variables for temporary needs that are confined to the current handler. To create a local variable, pick an arbitrary name, and assign a value to it. Variables such as `i`, `j`, and `k` are commonly used for loops or indices. Variables such as `x` and `y` are commonly used for coordinates.

In this example, `i` and `y` are local variables (everything else is a reserved Lingo keyword).

Example 1-9: Using Local Variables

```
on mouseUp
  set y = the locV of sprite the currentSpriteNum
  repeat with i = 1 to 100
    set the locV of sprite the currentSpriteNum = y + i
    updateStage
  end repeat
  put i
  put y
  showLocals
end
```

Local variables are "private" to the handler in which they are used. The *showLocals* command must be used from *within* the handler for which you wish to display local variables. Likewise, you cannot use *put* from the message window to display a local variable.

Local variables are independent of other variables in other handlers that may have the same name. Because they cannot be used until they are assigned a value, local variables have no default value. Using one before assigning it a value results in a "*Variable used before assigned a value*" error. In this example, `x` is an uninitialized local variable and will cause an error.

```
on mouseUp
  if x = 5 then
    go frame 15
  end if
end
```

See "*Special Treatment of the First Argument Passed*" later in this chapter for an explanation of why using an undeclared local variable as the first argument to a function does *not* generate an error.

Parameters

Parameters are local variables that automatically receive the value(s) of incoming arguments used in the call to the handler (see "*Parameters and Arguments*" and "*Generalizing Functions*" later in this chapter). Parameters are *declared* (named) on the same line as the handler name.

Example 1-10: Using Parameters

```
on someFunction param1, param2, param3
  -- The && operator assembles the string for output
  put "The three input parameters are" && [LC]
        param1 && param2 && param3
end

someFunction 1, "b",  7.5
-- "The three input parameters are 1 b 7.5
```

Parameters can assume different values each time a handler is called. A different copy of the parameters is created each time the handler is called and disappears when the handler ends. Changes to parameters within a handler generally have no effect outside that handler, but modifying a Lingo list passed as a parameter will modify the original list in the calling routine as well. See Chapter 6, *Lists* for important additional details. Don't use the name of a global variable as the name for a parameter or other local variables. See "*Common Errors When Using Variables*" later in this chapter

Global Variables

Global variables (or simply *globals*) are declared using the *global* keyword, and they persist throughout all movies until your Projector quits. They come into existence when a handler that declares them as *global* first runs and can be accessed by any handler that also declares them as *global*. Global variables can be displayed in the Message window using *showGlobals* (the built-in global variable *version* always appears in the list of globals).

```
showGlobals
-- Global Variables --
version = "6.0.2"
```

Whenever you test or set a variable in the Message window it is treated as a global variable.

```
set anyVariable = 5
```

```
showGlobals
-- Global Variables --
version = "6.5"
anyVariable = 5
```

Use *clearGlobals* to reset all global variables to VOID, except for the *version* global, which cannot be cleared. *ClearGlobals* also clears *the actorList* of the current movie in D6.

```
clearGlobals
showGlobals
-- Global Variables --
version = "6.5"
```

Globals can be shared by MIAWs and the main movie. Any change to the value of a global variable is reflected *everywhere* it is used. For clarity, name your globals starting with a "g." Global variables default to VOID (they have no value until one is assigned), but they retain their value even when playback stops or a new movie is loaded (unless *clearGlobals* is called). Shockwave clears global variables if the browser issues a *Stop()* command.

Globals are necessary when you want a variable to outlive the handler in which it is used or to send information between two handlers that are not otherwise connected.

In this example, *gOne* and *gTwo* are global variables shared by two handlers.

Example 1-11: Using Global Variables

```
on startMovie
  global gOne, gTwo
  set gOne = "Hello"
  set gTwo = 7
end
```

This handler can be in a different script than *startMovie*:

```
on mouseUp
  global gOne, gTwo
  if gTwo = 7 then put gOne
end
```

Lingo globals are declared, by convention, at the top of a handler immediately under the handler name; declaring globals in the middle of a handler is allowed but discouraged. You can declare more than one global with the *global* keyword by separating the variables with commas, as shown above. You can instead use a new line for each global declaration, such as:

```
global gOne
global gTwo
```

Globals declared *outside* of a handler (so-called "*global*" globals) are treated as if they were declared within all subsequent handlers within the same script.

For example, if the two handlers above are in the *same* script, the *global* declarations could be moved *outside* the handlers themselves and placed at the top of the script:

```
-- These are "global" globals and can be used
-- by all handlers in this script cast member
global gOne, gTwo

on startMovie
   set gOne = "Hello"
  set gTwo = 7
end

on mouseUp
  if gTwo = 7 the put gOne
end
```

Property Variables

Property variables are declared using the *property* keyword and persist as long as the object of which they are a property exists. (See Chapter 12, *Behaviors and Parent Scripts*, if you are not familiar with object-oriented programming, or just ignore this section for now.)

Property variables are programmer-defined and should not be confused with the built-in Lingo properties, although both are attributes of their respective objects.

Built-in Lingo properties are predefined attributes of built-in objects, such as sprites and cast members. Property variables are programmer-defined variables that are used to add attributes to their own objects (namely scripts).

Property variables are *instantiated* (created) when the parent script or Behavior that declares them is itself *instantiated* (either by Director or by the programmer). For example, when Director encounters a Behavior attached to a sprite in the Score it instantiates that Behavior and its properties. (This is explained in detail in Chapter 12. For now, just assume that when a Behavior is encountered in the Score, Director assigns appropriate values to any property variables that the Behavior declares.)

Properties can then be accessed by any handler within the same script. Each *instance* (use or occurrence) of the parent script or Behavior gets its own copy of the property variables that are *independent* of other copies (despite having the

same name), just as all sprites and cast members have independent properties named *width* and *height*.

Property variables are declared at the top of the parent script or Behavior before the first handler. You can declare multiple property variables, separated by commas, using one *property* statement, or you can use separate *property* statements for each property variable. Property variables default to `VOID` but are usually assigned a value in a parent script's *new()* handler (or in a Behavior's *getPropertyDescriptionList()* handler; see Chapter 12). When a handler inside a parent script or Behavior is called, its private copies of those property values are used, unlike global variables that are shared among all scripts. For clarity, name your properties starting with a "p."

Property variables *must* be declared *outside of* any handlers in the script, as shown in the example below.

In this example, *pOne* and *pTwo* are property variables shared by the *new()* and *showProps* handlers, which are both presumed to reside in the same parent script cast member named "ParentScript."

Example 1-12: Using Property Variables

```
property pOne, pTwo

on new me, a, b
  set pOne = a
  set pTwo = b
  return me
end new

on showProps me
  put "pOne is" && pOne
  put "pTwo is" && pTwo
end showProps
```

To test it in the Message window, first instantiate the parent script by calling the *new()* handler. When the programmer instantiates a script, he customarily specifies initial values for the properties by specifying them as arguments to the *new()* handler (although some programmers prefer to assign properties as a separate step from instantiating a script). We create one instance using the integer 6 and the string "hello," for the properties *pOne* and *pTwo*. We then create a second instance with different values to be used as *pOne* and *pTwo*.

```
set instance1 = new (script "ParentScript", 6, "hello")
set instance2 = new (script "ParentScript", 9, "goodbye")
```

We pass an *instance* created using *new()* to *showProps* (either `instance1` or `instance2`). That allows *showProps* to determine the correct properties for each

instance separately. Note how the results printed by *showProps* depend on which instance variable we pass to it.

```
showProps (instance1)
-- "pOne is 6"
-- "pTwo is hello"
showProps (instance2)
-- "pOne is 9"
-- "pTwo is goodbye"
```

Note that inside the script that declares them, property variables are accessed by using their name, as shown in *showProps* above. Programmer-defined property variables can also be accessed from *outside* a script instance using the script instance and the keyword *the,* such as:

```
put the propertyVariable of scriptInstance
```

Property variables belonging to scripts can also be accessed by referencing the script itself rather than an instance of the script, such as:

```
put the propertyVariable of (script "myScript")
```

Although they can also be accessed using the keyword *the*, remember that property variables are programmer-defined and are not the built-in Lingo properties that also happen to start with the keyword *the*.

Continuing the example above, you can access the properties of *instance1* and *instance2* without using *showProps*, instead using:

```
put the pOne of instance1
-- 6
put the pTwo of instance2
-- "goodbye"
```

Note that simply typing *put pOne* or *put the pOne* in the Message window would fail because you must specify the script instance that owns the property. If no script instance is specified, the property name must be a built-in system property, such as:

```
put the colorDepth
-- 8
```

Refer to Chapter 12, especially if this section left you thoroughly confused.

Common Errors When Using Variables

The following are the most common errors of both new and experienced programmers when using variables.

Using a variable before assigning a value to it:

Attempting to use a variable that has never been declared or assigned a value causes a "*Variable used before assigned a value*" error, such as:

```
set the locV of sprite 1 = y
```

(This won't fail in the Message window because *y* will be treated as global containing the value VOID, but it will fail from within a script.)

New local variables must first be assigned a value:

```
set y = 5
set the locV of sprite 1 = y
```

Global variables can be declared with the *global* keyword, without necessarily being assigned a value (we presume the global was assigned a meaningful value elsewhere; if not, it defaults to VOID):

```
global gLocForSprite
set the locV of sprite 1 = gLocForSprite
```

In the following statement, Director complains about the undeclared local variable *y*, which we are attempting to use in the expression although it has not been previously assigned a value. Director does *not* complain about the new local variable *x*, to which we are attempting to assign a value. In other words, the right side of the equation can use only existing variables, but the left side of the equation can be either an existing or a new variable.

Not knowing which storage class to use for a variable:

Use property variables for attributes that have a different value for each instance of a parent script or Behavior or that must persist for the life of the object. Use global variables when a value must outlive the handler, object, or movie in which it is used or must be accessible to multiple handlers, objects, or movies. Use local variables for values that are used only for convenience within the current handler and then discarded, such as indices in a repeat loop or interim steps in a mathematical calculation. Use parameters to accept inputs that can make a handler more flexible (see "*Parameters and Arguments*" later in this chapter).

```
set x = y
```

Not knowing when to use a variable:

Most novices use either too many or too few variables. Use variables whenever you want Director to remember something, such as the results of calculations, user input, lists of items, or anything that you need more than once. Programming is like cooking. You may be able to cook dinner in one pot, or you may need two frying pans and a pressure cooker; it depends on the recipe and your personal style. This example:

```
on mouseDown
  if the locH of sprite (the currentSpriteNum) > 50 then
    put the locH of sprite (the currentSpriteNum)
  end if
end
```

could be rewritten as:

```
on mouseDown
  set myLocH = the locH of sprite (the currentSpriteNum)
  if myLocH > 50 then
    put myLocH
  end if
end
```

Both examples are equivalent, but the second one is somewhat easier to read and maintain because the result of the lengthy expression is stored in *myLocH*, which is then used for comparing and displaying the value.

Using a "stale" value stored in a variable, instead of the current value:

When you assign a variable, it records a snapshot in time. You must recalculate values that are expected to change. The following is wrong because *y* never changes after its initial value is assigned:

```
set y = the mouseV
repeat while y > 50
  put "The mouseV is greater than 50"
end repeat
```

Instead, check the current value of *the mouseV* property repeatedly:

```
repeat while the mouseV > 50
   put "The mouseV is greater than 50"
end repeat
```

Forgetting the keyword the when using a Lingo property name:

Why would *mouseV* cause a "*Variable used before assigned a value*" error?

```
repeat while mouseV > 50
   put "The mouseV is greater than 50"
end repeat
```

Compare the above *repeat...while* statement to the previous example.

Using clearGlobals carelessly:

ClearGlobals indiscriminately resets all global variables to `VOID`. (In D6, it also sets *the actorList* to `[]`.) This will make any Lingo code that relies on global variables or *the actorList* lose whatever it had stored in them. This is usually a very bad thing, and it can be hard to track down if you are working with multiple programmers or large projects. Set individual globals to `VOID` to clear them separately instead.

Using the same variable name as both a global and local variable:

The most common error is to declare a variable *global* in one handler and forget to declare it *global* elsewhere.

In that case, it is implicitly a separate local variable in the second handler, despite having the same name as a global in the first handler, such as shown in Example 1-13.

Example 1-13: Common Errors with Global Variables

```
on initGlobal
  global gMyValue
  set gMyValue = 27
end initGlobal

on readGlobal
 -- This causes a syntax error
  put gMyValue
end
```

The second routine must also declare *gMyValue* as a global:

```
on readGlobal
  global gMyValue
  put gMyValue
end
```

Test it from the Message window:

```
initGlobal
readGlobal
    -- 27
```

Here the error is reversed. The programmer forgot to declare *gMyValue* as a global in the first handler.

```
on initGlobal
   -- gMyValue is treated as a local variable.
   -- Setting it has no effect outside the handler.
   set gMyValue = 27
end initGlobal

on readGlobal
  global gMyValue
  -- The global, also named gMyValue, defaults to VOID
  put gMyValue
end
```

Test it from the Message window:

```
initGlobal
readGlobal
-- Void
```

Using a global incorrectly as a parameter:

It is acceptable, even common, to pass a global as an argument to a function (see "*Parameters and Arguments*" later in this chapter). In the receiving function, however, you must decide whether you intend to modify the global or merely use the global's value for a local operation.

In Example 1-14, *gUserTries* is a *local* variable within *gameOver()*. The global of the same name in *finishGame* will *not* be incremented.

Example 1-14: Passing Globals Variables as Parameters

```
on finishGame
  global gUserTries
  gameOver (gUserTries)
end finishGame

on gameOver gUserTries
  set gUserTries = gUserTries + 1
end gameOver
```

This can be rewritten in one of two ways. Here, *gUserTries* is declared *global* within *gameOver()*:

```
on gameOver
  global gUserTries
  set gUserTries = gUserTries + 1
end gameOver
```

Alternatively, a local variable can be used and passed back to *finishGame*.

```
on finishGame
  global gUserTries
  set gUserTries = gameOver (gUserTries)
end finishGame

on gameOver inValue
  return (inValue + 1)
end gameOver
```

La Persistencia de la Memoria

Recall that each time a handler is called, any local variables are discarded when the handler finishes. If you want a variable's value to persist over repeated calls to a handler, you must declare it as a global or property variable. (See Chapter 12 for details on property variables that persist for the life of the object that declares them.)

Example 1-15 counts the number of times that *countMe* is called. Don't forget to reset the global as needed, as shown in *testCount*.

Example 1-15: Persistent Variables

```
on countMe
  global gCounter
  set gCounter = gCounter + 1
  put "This handler has been called" && gCounter && "time(s)"
end countMe

on testCount
  global gCounter
  set gCounter = 0
  repeat with i = 1 to 10
    countMe
  end repeat
end

testCount
-- This handler has been called 1 time(s)
-- This handler has been called 2 time(s)
-- etc.
```

Property variables in *score* scripts (that is, Behaviors) also persist over time (but you can not use them with *castmember* scripts):

```
property gCounter
on mouseUp me
  if gCounter > 10 then
    alert "Are you tired of clicking yet"
  else
    set gCounter = gCounter + 1
  end if
 end mouseUp
```

One-Time Initialization

You may wish to perform some initialization once and only once. You can use a global variable, as shown in Example 1-16, to track whether the initialization has taken place.

Example 1-16: One-Time Initialization

```
on startMovie
  global gBeenDone, gUserScore
  -- If the global is VOID, we haven't initialized yet
  if voidP(gBeenDone)
    -- Do initialization here
set gUserScore=0
    -- Set the global flag to indicate its been done
    set gBeenDone = TRUE
  end if
end startMovie
```

Variable-Related Lingo

Table 1-1 shows the Lingo commands related to variables, including declaration, assignment, instantiation, and deallocation of various variable types.

Table 1-1: Variable-Related Commands

Keyword	**Usage**
ancestor	Reserved name for declaring an ancestor property. See Chapter 12.
birth()	Obsolete, previously used to instantiate an object. See *new*.
clearGlobals	Resets all global variables, except *version*, to `VOID`. Also clears *the actorList* in D6. Don't use this when working with other programmers who rely on global variables or `the actorList`.
global *gName1* {, *gName2*}	Declares one or more global variables.
global version	Declares the reserved global *version*.
list(), [], or [:]	Allocates a list. See Chapter 6.
mNew	Used to instantiate an XObject. See Chapter 13, *Lingo Xtras and XObjects*.
mDispose	Used to deallocate an XObject instance. Chapter 13.
new (*script*, *args*) new (xtra "*XtraName*")	Creates a child object or Xtra instance.
param(*n*)	Indicates the value of the n^{th} parameter received by a handler. See *the paramCount*.
the paramCount	Indicates the number of parameters received by a handler. Use it to check how many arguments were specified in the function call.

Table 1-1: Variable-Related Commands (continued)

Keyword	Usage
property *pName1*{, *pName2*}	Declares one or more property variables.
property ancestor	Declares an ancestor property. See Chapter 12.
put *variable*	Prints a variable's value in the Message window.[1]
put *value* into *variable*	Assigns a value to a variable. I prefer using the *set* command.
set *variable* = *value* set *variable* to *value*	Assigns a value to a variable. See also *put … into*
showLocals	Prints all local variables in the current handler. Must be used from within a handler, not from the Message window.
showGlobals	Prints all global variables in the Message window.
Stop()	Stops a Shockwave movie from within the browser and clears global variables.
the *property* {of *object*}	Refers to a built-in Lingo property.
the *property* of *instance*	Refers to a programmer-defined property variable of a parent script or Behavior.
the *property* of *list*	Refers to properties within a property list. See Chapter 6.
version	A global variable containing Director's current version. Use *the productVersion property* to determine the version without declaring a global.

[1] Use *put* from within a handler to display the value of any variable (including local variables) within that handler in the Message window. Global variables can be tested using the *put* command in the Message window iitself. Use the *alert* command to display the value of variables in a dialog box, especially from within a Projector where the *put* command has no effect.

Allocation and Scope of Variables

Local variables in Director are allocated *implicitly* (that is, without your explicit instruction) when you assign a value to them. Global and property variables are allocated explicitly using the *global* and *property* keywords. The amount of memory variables require depends on what is stored in them. Local variables are deallocated automatically when a handler terminates. Properties are deallocated when the object of which they are a property is disposed.

The main concern is objects that you never dispose and global variables (which persist indefinitely) that refer to items that require a lot of memory, such as long lists or strings. Global variables are never truly disposed, but you can reduce the memory they use by assigning them to zero or `VOID`. (Avoid using *clearGlobals* for this purpose, as it resets all global variables indiscriminately and in D6 it also clears *the actorList.*) Objects are disposed by setting all variables that refer to the

object to zero or `VOID`. See Chapters 12 and 13. Table 1-2 shows the scope of individual variables of the different data storage classes.

Table 1-2: Data Storage Class Scope

Manner in Which Variable Is Declared	**Variable's Scope**
No explicit declaration; variable is implicitly local	Can be used in current handler, but only after it is assigned a value, using *set...=* or *put...into*.
Declared as arguments in the handler definition	Implicitly local, but automatically initialized to value of incoming arguments. Same scope as a local variable. See "*Parameters and Arguments*."
Assigned a value in Message window	Variables assigned in the Message window are implicitly *global*. They must still be declared global within scripts that wish to make use of them.
Explicitly declared as a global inside a handler	Variable is *global* only within handler in which it is declared.
Explicitly declared as a global `outside` of any handler, at the top of a script	Variable is *global* only within handlers in the same script cast member *after* the point at which it is declared.
Explicitly declared as a property within a handler	Not supported.
Explicitly declared as a property outside of a handler, at the top of a parent script or Behavior	Property variable is accessible within any handler in the parent script or Behavior. Its scope is limited to the current instance of the script. Outside the script, it can be accessed with *the property of scriptInstance*.
Explicitly declared as a property within an ancestor script	Property variable is accessible within the ancestor script or to any object using that script as an ancestor
Do statement	Scope is only the current *do* statement. See "*Do Statements*," under "*Dynamic Script Creation*."

Lingo's Skeletal Structure

This section covers the bones on which you'll hang the flesh of your Lingo program. A handful of Lingo commands and functions control program structure and execution. Each line of Lingo is executed in order, unless the Lingo itself changes the flow. Lingo detours to execute any handler names it encounters, as shown in detail under "*Nested Handlers*" in the earlier section, "*Handlers and Scripts*." A detour taken to execute another handler is known as a *subroutine call* in most languages. Refer to Chapter 1, *How Director Works* in *Director in a Nutshell* regarding the difference between Lingo's program flow and the movement of the Score's playback head.

Comments

Although Lingo tells Director what to do, the Lingo code is ultimately written and maintained by humans (or programmers, anyway). Lines starting with two hyphens (--) are comments that Director ignores; they do not increase the execution time. Adding comments to someone else's code is a great way to make sure you understand what it does.

I place comments *before* the code they describe, although some people place them afterward, which seems patently absurd to me. Comments should give a preview, not a postmortem.

Always describe the big picture:
: Include comments at the top of your handler to describe its purpose. Describe *what* the handler does and *why*. Describe the data stored in major variables, such as lists, and which global variables the code uses.

Use comments liberally, even when you are the only programmer:
: Your own comments will help you immensely if you have to fix a bug six months later, or they'll help the next poor slob who has to maintain your code when you take a better job.

Write comments that complement, not simply reiterate, the programming code:
: Your comments should explain what the code is doing at the conceptual level, not the syntactical level. A comment such as "*Set x to zero*" is useless. Use informative comments like "*Reset the user's game score.*" Assume that the person reading your comments understands Lingo's basic operation. Don't document how Lingo itself works; document what *your* code accomplishes.

"Comment out" temporary changes or tests:

gComments can exist on separate lines or at the end of a line following other Lingo commands. Anything following a comment character will be ignored until the next carriage return. Lingo does not have a way to create multiline comments, as in C, but you can use the continuation character or use more double-hyphens on subsequent lines.

Example 1-17: Comments

```
-- This is a Lingo comment on its own line

set x = 1  -- This is a Lingo comment at the end of a line

-- Here's a "commented out" command that is ignored
-- set x = 1

-- To create a multiline comment in Lingo,
-- begin each line with its own comment delimiter

-- This is a multiline comment by virtue of the ¬,
continuation character at the end of the previous line

-- Beware!  The last line is part of the comment
-- by virtue of the continuation character ¬
set x = 1
```

This example demonstrates appropriate comments.

```
on resetHighScores
  -- This handler clears the high score chart
  global gHighScoreList

  -- Reset the list holding the top scores
  -- and the winners' names
  deleteAll gHighScoreList

  -- Clear the on-screen high score winners chart
  set the text of field "HighScore" = EMPTY

  -- Reset the sprites displaying the highest score
  repeat with i = 12 to 20
    set the member of sprite i = member "BigZero"
  end repeat
end
```

To comment or uncomment your code:

Manually add or delete two hyphens (--) at the beginning of one or more lines of code.

Use the *Comment* and *Uncomment* buttons in the Script window's button bar (see Figure 3-3) to comment or uncomment the currently highlighted line(s).

Use `Cmd-Shift->` and `Cmd-Shift-<` (Mac) or `Ctrl-Shift->` and `Ctrl-Shift-<` (Windows) to comment or uncomment the currently highlighted line(s). (These are available under Director 4's `Text` menu.) Use the Comment and Uncomment options under the context-sensitive pop-up menu in the Script Window, Ctrl-click (Mac) or right click (Windows).

A good rule of thumb is one comment for every three lines of Lingo code. Use self-explanatory variable, symbol, and handler names to improve your code's readability.

You can use the Lingo command *nothing* as a placeholder when you don't want to execute any commands. Unfortunately, unlike comments *nothing* takes time for Lingo to "execute."

Multi-line Code Structures

Lingo commands generally occupy a single line (that is, they end when a carriage return is encountered), but Table 1-3 shows Lingo statements that occupy more than one line. Each multi-line command begins with its own special keyword and is terminated by some variation of the *end* keyword. Director automatically indents the statements within the body of a multi-line command two additional spaces. If the indentation is wrong, something is wrong with your Lingo. The Lingo continuation character ¬ should only be used to break long lines, not to "join" the multiple lines that are part of a multi-line structure itself.

The multi-line code fragments shown as examples must be incorporated into a handler for testing; they will not work from the Message window.

Table 1-3: Multi-line Code Structures

Structure	**Usage**
¬ Created with `Option-Return` (Mac) or `Alt-Enter` (Windows)	Continues Lingo onto next line.

`on handlerName`
`{parameter1,parameter2,…}`
`statements`
`end {handlerName}`[1]
	Handler declaration.

`if expression then`
`  statements`
`else if expression then`
`  statements`
`else`
`  statements`
`end if`
	Executes Lingo conditionally depending if expression is TRUE or FALSE. See "*If...Then Statements*" later in this chapter and see Chapter 5, *Data Types and Expressions*.

`case (expression) of`
`value1:`
`  statements`
`value2, value3:`
`  statements`
`otherwise:`
`  statements`
`end case`
	Executes Lingo when an expression matches one of the specified values (see "*Case Statement*" later in this chapter)

`beginRecording`
`  statements`
`endRecording`[2]
	Score recording.[3]

`tell`
`  statements`
`end tell`
	Window messaging.[4]

`repeat while expression`
`statements`
`end repeat`
	Repeats until expression becomes `FALSE`.

`repeat with item in list`
`statements`
`end repeat`
	Cycles once through all the items in a list.

Table 1-3: Multi-line Code Structures (continued)

Structure	Usage

`repeat with index = start to end`
`statements`
`end repeat`
	Repeats for a range of values increasing by 1.

repeat with *index* = *last* down to *first* *statements* end repeat	Repeats for a range of values decreasing by 1.

[1] The *handlerName* following the *end* keyword is optional. Include it especially when concluding long handlers whose entirety is not visible in the Script window.
[2] Note that *endRecording* is one word whereas the other *end* commands are two words.
[3] See Chapter 3, *The Score and Animation*, in *Director in a Nutshell*.
[4] See Chapter 6, *The Stage and Movies-in-a-Window*, in *Director in a Nutshell*.

Common Handler Declaration Errors

All handlers start with the keyword *on*, followed by the handler name, and end with the keyword *end*. A script can contain multiple handlers, one after the other. Always end one handler before beginning the next one. Let Director's auto-indentation be your guide.

Example 1-18: Handler Declarations

This is wrong:

```
on mouseUp
  statements

on mouseDown
   statements
end
```

This is right:

```
on mouseUp
   statements
end mouseUp

on mouseDown
   statements
end mouseDown
```

Note the optional use of the handler name after the *end* keyword. This is useful for making sure that you always have a matching *end handlerName* statement for each *on handlerName* statement. A handler declaration can also include parameters (not shown). Refer to "*Parameters and Arguments*" later in this chapter.

Common Multi-Line Structure Errors

Always terminate nested multiline structures from the inside out, as with nested parentheses.

Note the incorrect indentation in Example 1-19 caused by the *end if* incorrectly preceding the *end repeat*, and note how the multi-line structures incorrectly cross, rather than nest.

Example 1-19: Nested Multi-Line Structure Errors

Wrong

```
on mouseUp
if the clickOn = 3 then
repeat with x = 1 to 7
  puppetSprite x, TRUE
end if
end repeat
end
```

Note the corrected indentation, and see how the multi-line structures nest neatly, rather than cross.:

Right

```
on mouseUp
  if the clickOn = 3 then
  repeat with x = 1 to 7
    puppetSprite x, TRUE
  end repeat
  end if
end
```

Don't use a continuation character ¬ improperly at the end of a line. Director automatically knows that the multiline command continues on the next line.

The following is *wrong*. You should not use the ¬ character in this case:

```
repeat with x = 1 to 7 ¬
  puppetSprite x, TRUE
end repeat
```

Here is an example of *correctly* using an ¬ character within the *body* of a multi-line code structure to break a long line of Lingo.

```
repeat with x = 1 to 7
  set the member of sprite x = ¬
  member "Pinky and the Brain" of castLib "World Domination"
end repeat
```

Conditional Execution

You can have Director execute different Lingo code in response to various conditions. Any non-zero expression (including negative numbers) is considered `TRUE`, and only an expression that evaluates to zero is considered `FALSE`. Refer to Chapter 5 for details on evaluating comparisons, including compound expressions. (These example multi-line code fragments must be placed in a handler for testing.)

If...Then...Else...End If Decisions

The *if* statement will execute different Lingo code based on the value of the specified expression. This allows you to, say, use a single Director movie on both Macintosh and Windows but branch to different code depending on the playback platform, as determined by *the platform* property, such as:

```
if (the platform starts "Windows") then
  -- Do Windows-specific stuff here
else
  -- Do Macintosh-specific stuff here
end if
```

The *if* statement has many possible forms, but I recommend only these. Items shown in curly braces are optional. Note the correct indentation:

Single-Clause If...Then...End If

```
if expression  then
  statement1
  statement2
end if
```

Multiple-Clause If...Then...Else If...Else...End If

The *else if* and *else* clauses are optional:

```
if expression  then
   statement1
{else if expression  then
   statement2}
{else
   defaultAction}
end if
```

For example:

```
if x = 1 then
  go frame "Slide1"
else if x = 2 then
  go frame "Slide2"
else if x = 3 then
  go frame "Slide3"
else
  go frame "End"
end if
```

One-Line and Two-Line If Statements

The following forms are theoretically valid, but they often misbehave in Projectors and within nested *if* statements, so I don't recommend them:

```
if expression  then statement1
```

or

```
if expression  then statement1
else statement2
```

The Classic Three-Line If Statement

The one-line and two-line forms of the *if* statement make it hard to tell if the indenting is correct even after Director auto-formats your Lingo.

Where I use the one-line form of *if...then* in the examples throughout the book, I do so only for brevity. The Lingo compiler occasionally makes errors when evaluating one-line and two-line *if* statements.

Use the following three-line form even for simple *if* statements, *especially* when nesting *if* statements.

```
if (expression) then
  statement
end if
```

By always using an *end if* statement, and by always including a carriage return after the *then* keyword, Director won't get confused and neither will you. The auto-indenting will work reliably and it also makes the code easier to read and debug. See Chapter 3.

Nested Ifs

You can nest *if* statements to create the desired logic. It is crucial that you use the three-line *if...end if* construct, rather than the one-line or two-line form. An *end if* always pairs up with the most recent *if* from the inside out. Let Lingo's auto-indentation be your guide.

```
if expression1 then
  if expression2 then
    action1
  else
    action2
  end if
else
  defaultAction
end if
```

Example 1-20 is a typical usage of a nested *if* statement:

Example 1-20: Nested If Statements

```
if field "name" <> EMPTY then
   -- These five lines of code are a nested "if"
   if length(field "name") > 10 then
      alert "Enter only the first ten letters of your name"
   else -- This "else" pairs with the second "if"
      alert "Welcome to my nightmare" && field "name"
   end if --This "end if" pairs with the second "if"

else -- This "else" pairs with the first "if"
   alert "Please enter your name in the field"
end if --This "end if" pairs with  the first "if"
```

Common Errors When Using If Statements

Although conceptually simple, the *if* statement consistently confuses new Lingo programmers. Avoid the common pitfalls shown in the following examples.

Omitting the End If or the Then

There must be one *end if* for each *if* statement. Watch for this, especially with nested *ifs*, such as is shown in Example 1-21.

Example 1-21: Common If Statement Errors

```
if x > 5 then
  if x < 7
    statements
  end if
-- You are is missing an "end if" here
```

If you omit the *then* keyword, you'll also have problems:

```
if x > 5     -- the keyword "then" is missing
  statements
end if
```

Failure to Use Else If

Use one *if...else if...end if* statement instead of a series of *if* statements.

The following is inefficient because if *x* equals 5, it will never equal 6, and vice versa:

```
if x = 5 then
  -- Do something
end if
if x = 6 then
   -- Do something different
end if
```

This is more efficient (see also "*Case Statements*" later in this chapter):

```
if x = 5 then
  -- Do something
else if x = 6 then
   -- Do something different
end if
```

Improper Comparison Expressions

The incorrect order of comparison can lead to the wrong code being executed. Some code may never be executed, or code may be executed unintentionally.

In this example, *both if* statements will be executed if ***x*** equals 5. This may be what you want, but it is more likely a logic error:

```
if x > 0 then
  statements
```

```
  end if
   if x > 4 then
      statements
   end if
```

In this erroneous example, the second branch will *never* be executed because the first branch is taken in every case where the second condition would be `TRUE`:

```
   if x > 0 then
     statements
   else if x > 4 then
      alternative statements (never reached)
   end if
```

In this corrected example, the second branch will be executed *only* if ***x*** is less than 4 but greater than 0:

```
   if x > 4 then
     statements
   else if x > 0 then
      alternative statements
   end if
```

Nesting If Statement Unnecessarily or Incorrectly

Nested if statements seem to give people fits. Don't use a nested *if* statement when an *if...then...else if...end if* statement will suffice.

The innermost conditional clauses in this example are never executed.

Example 1-22: Nesting If Statements Properly

```
if x = 3 then
  alert "Executed option 3"
  -- This is never executed; if x is 3, it's not 4
  if x = 4 then
    alert "Executed option 4"
    -- This is never executed; if x is 4, it's not 5
    if x = 5 then
      alert "Executed option 5"
    end if
  end if
end if
```

The correct construct is:

```
   if x = 3 then
     alert "Executed option 3"
   else if x = 4 then
     alert "Executed option 4"
else if x = 5 then
  alert "Executed option 5"
   end if
```

Excess End If Statements

Use an *end if* only for each *if*, not for each *else if* (sometimes Director won't complain, but the results won't be what you wanted). Example 1-23 is *wrong*.

Example 1-23: Using End If Improperly

```
if x = 3 then
  alert "Executed option 3"
else if x = 4 then
  alert "Executed option 4"
else if x = 5 then
  alert "Executed option 5"
end if
end if   -- This is not needed
end if   -- This is not needed
```

Extraneous *end if*s can change your logic unintentionally. Contrast the following with the earlier nested *if* example. It will always execute the last *alert* command because it follows the last *end if*:

```
    if field "name" <> EMPTY then
      -- These five lines of code are a nested "if".
      if length(field "name") > 10 then
        alert "Enter only the first ten letters"
      else
        alert welcome & field "name"
      end if   -- This terminate the inner "if" statement
    end if -- This terminate the outer "if" statement
    -- This will always be executed
    alert "Please enter your name"
```

Missing End If Statements

Each *if* statement must have a matching *end if*. Erroneous structures at the end of a preceding handler will trickle into the next handler and corrupt the indentation. Note the incorrect indentation in the *mouseDown* handler caused by the missing *end if* in the preceding *mouseUp* handler:

```
    on mouseUp
      if the clickOn = 3 then
        if rollover(7) then
          put "Yahoo!"
        end if
        put "whatever"
        -- This is missing an "end if"
    end

on mouseDown
    put "Hello"   -- Note incorrect indentation
    end
```

Note the corrected indentation of *put "Hello"* in the *mouseDown* handler:

```
on mouseUp
   if the clickOn = 3 then
     if rollover(7) then
       put "Yahoo!"
     end if
     put "whatever"
   end if   -- That's better!
 end

 on mouseDown
   put "Hello"
 end
```

Inefficient Use of If Clauses:

Look for ways to reduce the number of *if* clauses. Here we've modified the example shown under "*Multiple-Clause If...Then...Else If...Else...End If.*" We've reduced the number of clauses by using the value of *x* to construct the name of the destination marker ("Slide1," "Slide2," or "Slide3").

```
if x >= 1 and x <= 3 then
  go frame ("Slide" & x)
else
  go frame "End"
end if
```

Case Statements

The *case* statement conditionally executes Lingo statements based on the value of an item. It is often easier to implement than multiple *if...then* statements, but long *case* statements can be slower than the corresponding *if...then* constructs. Director executes the statement(s) following the first *value* that matches the *case* clause. The colon after *otherwise* is optional, and multiple statements can be included after each value to be matched. Once a match is found and its statements executed, subsequent values and their statements are ignored, and execution continues after the *end case* statement.

```
case (item)of
  value1:
    statement
  value2:
    statement
    statement
  value3, value4:
    statement
  otherwise:
    statement
end case
```

Example 1-24 shows a *case* statement and the equivalent *if* statement.

Example 1-24: Case Statements

```
on keyDown
  case (the key) of
    RETURN: go frame "done"
    TAB, BACKSPACE: beep
    otherwise: pass
  end case
end keyDown
```

The *case* statement is equivalent to the following *if… then* statement:

```
    on keyDown
      if (the key = RETURN) then
        go frame "done"
      else if (the key = TAB) or (the key = BACKSPACE) then
        beep
      else
        pass
      end if
    end keyDown
```

To use a comparative expression to branch within a *case* statement, use `TRUE` in the *case* clause, and enclose the comparative expression in parentheses, such as:

```
on keyDown
  case (TRUE) of
    (the keyCode >= 123 and the keyCode <= 126):
      -- The user pressed an arrow key
    statements
    (the keyCode = 122):
      -- The user pressed F1
    statements
    (the keyCode = 118):
      -- The user pressed F4
    statements
    otherwise:
      alert "Please press an arrow key, F1 or F4"
  end case
end keyDown
```

Repeat Loops

Repeat loops repeat any statements within the body of the loop. They are used to cycle through a series of items, such as elements in a list, or to repeat an operation a specific number of times (they are equivalent to so-called *for...next* loops used in some languages). When the *end repeat* command is reached, execution begins again at the top of the *repeat* loop until some condition causes the loop to terminate. Execution continues at the statement following the *end repeat* command. Use the Debugging window, described in Chapter 3, to examine the exact flow of Lingo as Director executes the steps in the *repeat* loop.

Most repeat loops are very fast, even for hundreds or thousands of iterations, but Director can't do anything else while you are executing a repeat loop, especially in Shockwave.

Don't use repeat loops that monopolize Director's attention for more than a few seconds.

In Shockwave, you can't check whether an operation completed with *netDone()* from within a repeat loop because Director doesn't perform network operations during a repeat loop. Likewise, Director doesn't update all system properties or update the Stage automatically during a repeat loop.

The repeat loop has four forms (3 forms of *repeat with* plus *repeat while*), as shown earlier in Table 1-3. The example multi-line code fragments must be placed in a handler for testing.

The *repeat while* command repeats as long as an expression is `TRUE`. If the expression never becomes FALSE, it will be an infinite loop.

Example 1-25: Repeat Loops

```
repeat while (the stillDown = TRUE)
  put "The mouse is still down"
end repeat
```

The *repeat with* commands loop through a range of values. Although not necessarily apparent, the three forms of the *repeat with* loop all use an *index variable* (counter) that changes automatically each time the loop is executed.

In this example, the index variable (`i`) starts at the initial value (in this case 1) and increments each time through the loop until it hits the upper bound (in this case 100). If the initial value is greater than the upper limit, the statement(s) within the loop are never executed.

```
repeat with i = 1 to 100
   put "The next number is" && i
end repeat
```

The following is the equivalent *repeat while* loop to the above *repeat with* loop. You can see that *repeat with* loops are more convenient.

```
set i = 1
repeat while i <= 100
  put "The next number is" && i
  set i = i + 1
end repeat
```

The *repeat with...down to* command repeats for a decreasing range of values. In this case, the index variable (`i`) is decremented (not incremented) each time through the loop.

```
repeat with i = 100 down to 1
```

```
  put string(i) && "Bottles of beer on the wall..."
end repeat
```

The *repeat with...in* command cycles once through all the items in a list. (Refer to Chapter 6 or ignore this example for now.) Each time through the loop, `i` is automatically set to the next item in the list. Use the following to loop through an irregular set of numbers:

```
set myList = [12, 17, 52, 43]
repeat with i in myList
  put "The next item in the list is" && i
end repeat
```

The variable `i` is *not* an integer in the previous example; it is actually the contents of the next item in the list. The previous example can be simulated with a standard *repeat with* loop, as follows:

```
set myList = [12, 17, 52, 43]
repeat with j = 1 to count (myList)
  set i = getAt (myList , j)
  put "Item number:" && j
  put "The next item in the list is" && i
end repeat
```

In the last example, note that the index variable *j* is an integer and can be used to print a list element's position within the list. You must manually intialize and increment an index variable to obtain a similar counter if using a *repeat with...in* loop, such as:

```
set myList = [12, 17, 52, 43]
set j = 1
repeat with i in myList
  put "Item number:" && j
  put "The next item in the list is" && i
  set j = j + 1
end repeat
```

Altering Loop Execution

Use *next repeat* to skip the current iteration of a *repeat* loop and to continue with the next iteration.

```
repeat with x = 1 to the number of members
  if the memberType of member x = 0 then
    next repeat
  end if
  put "Item" && x && "is type" && the memberType of member x
end repeat
```

Use *exit repeat* to exit the current repeat loop immediately. Program execution continues with the statement following the *end repeat* statement. *Exit repeat* exits only the current (innermost) repeat loop, and it will not exit nested *repeat* loops. Use *exit*, or *abort*, or a flag to exit multiple loops, as shown in Example 1-26.

(Again, use the Debugging window to examine the exact flow of Lingo as Director executes the following code.)

Example 1-26: Nested Repeat Loops

```
on findIt
  global gFoundIt
  set gFoundIt = FALSE
  -- Search for a #shape cost member
  repeat with y = 1 to the number of castLibs
    repeat with x = 1 to the number of members of castLib y
      set thisItem = the memberType of member x of castLib y
      if thisItem = #shape then
        set gFoundIt = member x of castLib y
        exit repeat -- exit the innermost loop
      end if
    end repeat
   -- exit the outermost loop too
    if gFoundIt then
      exit repeat
    end if
  end repeat

  -- Execution continues here after loop
  if gFoundIt then
    put "Found shape cast member at" && gFoundIt
  else
    put "Shape cast member not found."
  end if
end
```

Manually Controlling the Loop's Counter

Lingo does *not* precalculate the number of iterations it will perform for a *repeat... with* loop. Rather, it reevaluates the expression each time through the loop. In the following example, we manually increment the index variable (`i`) to step by two rather than one. Note that we add only 1 to `i`, not 2, because Lingo will automatically increment `i` once each time the loop is executed.

Example 1-27: Customized Repeat Loops

```
on printEvenNumbers
  repeat with i = 0 to 100
    put i
    set i = i + 1
  end repeat
end printEvenNumbers
```

Avoid manually setting the index variable within a *repeat* loop unless you need to change the number of loop iterations. Setting it incorrectly can lead to an infinite loop.

After a loop completes, the index normally is one greater than the ending value:

```
on testRepeat
  repeat with i = 1 to 100
    put i
  end repeat
  put "The ending value for i is" && i
end testRepeat

testRepeat
-- 1
-- 2
-- <etc.>
-- 99
-- 100
-- "The ending value for i is 101"
```

Infinite Loops

An *infinite loop* is a *repeat* loop that will never be exited because the conditional expression never turns `FALSE`. (Apple's street address in Cupertino, California, is One Infinite Loop). An infinite loop will appear to hang the computer, and you must abort the Projector or halt the movie to stop it.

The most simple infinite loop is shown in Example 1-28.

Example 1-28: Infinite Loops

```
repeat while TRUE
  -- do something
end repeat
```

Unless you add an *exit repeat*, Lingo will never exit such a *repeat* loop:

```
startTimer
repeat while TRUE
  if the timer > 60 then
    exit repeat
  else
    -- do something
  end if
end repeat
```

But this would be better written as:

```
startTimer
repeat while the timer <= 60
  -- do something
end repeat
```

It is easy enough to create an infinite loop accidentally. The following will loop forever, assuming the loop takes less than 60 ticks to execute, because *startTimer* is *within* the *repeat* loop and keeps resetting the timer.

```
startTimer
repeat while the timer <= 60
  startTimer
  -- do something
end repeat
```

The following would lead to an infinite loop as well. The variable ***x*** is unintentionally set to 5; the programmer did not realize that ***x*** is also being used as the loop's index variable.

```
repeat with x = 1 to 10
  put "x" && x
  set x = 5
end repeat
```

A loop will also be infinite if some condition you expect to become `FALSE` remains `TRUE` forever. Suppose you are waiting for a sound to start:

```
puppetSound "mySound"
repeat while not soundBusy(1)
  nothing
end repeat
```

If the sound never starts (in this case, you need to add *updateStage* after the *puppetSound* command), Director will loop forever because *soundBusy(1)* will never become TRUE, so *not soundBusy(1)* will always remain `FALSE`.

Use the Debugger to diagnose infinite loops and similar problems. See Chapter 3.

Parameters and Arguments

Imagine you own a calculator with buttons that each perform a single complete operation—one button adds 5 plus 7, and another button adds 5 plus 8, and so on. It might be convenient for those limited operations, but the calculator could rapidly become unwieldy. (Similarly, Chinese pictographs are inconvenient for computer usage compared to an alphabet from which you can construct any word.) In reality, each button on a calculator actually represents either an *operand*, such as the number 5, or an *operation*, such as addition, allowing for many possible combinations with relatively few keys.

At the beginning of this chapter we used the *alert* command to display the string "Hello World" in a dialog box. The *alert* command accepts any text string as an *argument.* An argument is analogous to the operands used in the example of a calculator above, and the *alert* command is analogous to the addition button that performs some operation using the argument(s). The words *parameters* and *argu-*

ments are often used interchangeably to indicate inputs that are passed to a function on which it can operate. Strictly speaking, an *argument* is the item specified as part of the function call, and a *parameter* is the same item once it is received inside the function.

Just as the *alert* command can accept *any* string for display, we should strive to make our custom handlers flexible by using variables to represent values that can change. In contrast, using a fixed number, such as 5, or a fixed string, such as "Bruce," in your program is called *hardcoding* a value.

Instead of *hardcoding* specific values into a handler, *generalize* the function so that it can operate on whatever *arguments* are passed into it.

For example, a handler that finds a file should be flexible enough to find *any* file we ask it to find, rather than always looking for a particular hardcoded filename. Beginning programmers often create two or more copies of the same handler with only minor variations to accomplish nearly identical tasks. Instead, you should create a *generalized* (that is, flexible) version of the handler that accepts arguments (or *parameters*) to accommodate the differences. This is shown in detail in Example 1-31 and 1-32. Let's start with a discussion of how arguments are passed to a function.

Passing Arguments

A simple function performs an operation on the argument(s) passed into it. For example, let's define an *avg()* function that averages two numbers.

Example 1-29: A Handler That Accepts Parameters

```
on avg a, b
  return (a+b) / 2.0
end
```

The names *avg*, *a*, and *b* are chosen *arbitrarily*. Note that there is a space between the handler name (*avg*) and the first parameter (*a*) but that subsequent parameters (such as *b*) are separated by a comma from the previous parameter. The *return* statement sends the answer back to whoever called the function. Without the *return* statement, the answer *avg* calculates would never be known! (Forgetting to return a result is a very common error. If the result from a custom function returns `VOID`, you probably forgot the *return* statement).

Type the handler in Example 1-29 into a movie script, and test it from the Message window. The *put* command prints the result returned by *avg()*.

```
put avg (5,8)
-- 6.5000
```

The integers 5 and 8 are *arguments* that are operated upon by *avg()*. The arguments are separated by commas. The parentheses are required to obtain the value

returned by *avg()*, but parentheses are optional when calling a command that does not return a value, such as *alert*.

The first argument (5) is automatically assigned to the first parameter (*a*), and the second argument (8) is assigned to the second parameter (*b*). In this case, the order of the parameters does not affect the result, but in most cases the order of the parameters is crucial. For example, division is not reflexive: 5 divided by 8 would not be the same as 8 divided by 5.

If the number of arguments does not match the number of parameters, Director won't complain. It is up to you to ensure that the correct number of arguments is specified.

Modify the *avg()* handler to create *newAvg* as follows:

```
on newAvg a, b
  put "The first parameter, a, equals" && a
  put "The second parameter, b, equals" && b
  set answer = (a+b)/ 2.0
  put "The answer is" && answer
end
```

We've added a local variable, arbitrarily named *answer*, which is convenient for holding the value that is printed and then returned to the calling program. Whereas *a* and *b* are *implicitly* assigned to the arguments in the function call, *answer* is explicitly assigned a value using *set...=*.

There is *no* difference (as far as the *newAvg* handler can tell) if we pass integer *variables* as arguments to *newAvg* instead of the integer *literals* 5 and 8. Type each of these lines in the Message window, pressing RETURN after each:

```
set x = 5
set y = 8
put x
newAvg(x, y)
-- "The first parameter, a, equals 5"
-- "The second parameter, b, equals 8"
-- "The answer is 6.500"
```

We don't need to use the *put* command because *newAvg* displays the result itself using *put*, rather than returning an answer.

The parameters within *newAvg*, namely *a* and *b*, are still equated to 5 and 8, respectively. The *values* of the arguments *x* and *y*, not *x* and *y* themselves, are passed to *newAvg*. This is called *passing arguments by value*, rather than *by reference*.

Refer to "*Parameter Passing*," in Chapter 4 for more details on parameters passed by reference and to Chapter 6 for how this affects Lingo lists.

Generalizing Functions

NewAvg is a generalized handler that can average any two numbers passed into it (we'll see later how to make it accept any number of arguments to average).

Generalized Sprite Handlers

To perform an operation on a sprite you must refer to the sprite by its channel number (or an expression that evaluates to a channel number). For a sprite in channel 1, you might use:

```
on mouseUp
  set the foreColor of sprite 1 = random (255)
end
```

A beginner might create another script to attach to a different sprite in channel 2 as follows:

```
on mouseUp
  set the foreColor of sprite 2 = random (255)
end
```

Not only is this wasteful, but these scripts will fail miserably if you move the sprites to a new channel. Thankfully, Lingo provides several system properties that can be used to generalize a handler. When a script is attached to a sprite, *the currentSpriteNum* property always indicates the sprite's channel number. Therefore, we can replace the two separate scripts with a single sprite script that can be attached to any sprite in any channel and that will always work.

Example 1-30: A Simple Generalized Behavior

```
on mouseUp
  set the foreColor of sprite (the currentSpriteNum) = random (255)
end
```

Refer to Chapter 9 for details on *the currentSpriteNum*, *the spriteNum of me*, and *the clickOn* properties and how they can be used to generalize handlers for use with any sprite.

Generalizing a Function with Parameters

The following is a more sophisticated example of a generalized function, but the principle is the same (feel free to skip this section if it is confusing). Let's suppose you want to check if the file "`FOO.TXT`" exists.

You might write the code shown in Example 1-31 (see Chapter 14 for an explanation of this Lingo and a more robust example).

Example 1-31: A HardCoded Function

```
on doesFooExist
  -- Use the FileIO Xtra to try to open the file FOO.TXT
  set fileObj = new (xtra "FileIO")
  if objectP(fileObj) then
    openFile (fileObj, "FOO.TXT", 1)
```

Example 1-31: A HardCoded Function (continued)

```
    set result = status (fileObj)
    -- A result of 0 indicates success
    if result = 0 then
      alert "FOO.TXT exists!"
    else
      -- Print the error message in the Message window
      put error (fileObj, result)
      alert "FOO.TXT can't be found"
    end if
   -- Clean up after ourselves
    closeFile (fileObj)
    set fileObj = 0
  end if
end doesFooExist
```

Now suppose you want to check if a different file exists. Most beginners would duplicate this long block of code, then change the filename in the *openFile()* function call from "FOO.TXT" to their new filename.

Never duplicate near-identical long blocks of code. Your code becomes harder to debug—and much harder to change if you do find a bug. *Always* generalize the code into a utility function that you can add to your "tool belt" for future use.

Below we've created a generalized function. Note that it accepts a file name as a parameter. *The name that you specify gets substituted automatically for the* `fileName` *parameter and is used in the openFile command.* Note also that it *returns* either `TRUE` (1) or `FALSE` (0) to indicate whether the file was found. If it couldn't be found, you may want to return the error code that was obtained from the FileIO Xtra's *status()* call. (It is good practice to simply return some result or status and let the caller decide whether to post an alert message or do something else.) Beyond that, it is essentially the same handler as shown in Example 1-31. Try to make your utility code as non-intrusive as possible, and clean up after yourself. Note that we opened the file in read-only mode to avoid failing if the file was already open. We also closed the file when done to clean up after ourselves. Example 1-32 is primarily for illustration. The FileIO Xtra will search in the current folder if *the searchCurrentFolder* is `TRUE` (the default). It will also search the list of paths, if any, in *the searchpaths*. It may not work correctly with long filenames under Windows. Thus Example 1-32 is not completely robust.

Example 1-32: A Generalized FileExists Function

```
on fileExists fileName
  -- Use the FileIO Xtra to open the specified file
  set fileObj =  new (xtra "FileIO")
  if objectP(fileObj) then
    -- Open file with mode = 1 "read-only"
    openFile (fileObj, fileName, 1)
```

Example 1-32: A Generalized FileExists Function (continued)

```
    set result = status (fileObj)
    -- A status of 0 indicates success
    if result = 0 then
      set found = TRUE
    else
      -- Display the error for debugging
      put error (fileObj, result)
      set found = FALSE
    end if
    closeFile (fileObj)
    set fileObj = 0
    -- Return TRUE or FALSE to indicate if the file exists
    return found
  end if
end fileExists
```

Now we can easily determine if *any* file exists, such as:

```
    put fileExists ("FOO.TXT")
    -- 1
    put fileExists ("FOOPLE.TXT")
    -- 0
```

Or use it as follows:

```
    if fileExists ("FOO.TXT") then
     -- Do whatever I want to with the file...
      -- such as open and read it
    else
      alert "The file can't be found"
    end if
```

Using a Generalized Function

Let's revisit the simple *newAvg* handler. Add the following handler to your movie script that already contains the *newAvg* handler shown earlier.

Example 1-33: Using Generalized Functions

```
on testAvg
  newAvg (5, 3)
  newAvg (8, 2)
  newAvg (4, 4)
  newAvg (7, 1)
end
```

Choose `Control➤Recompile Script`, and then test *testAvg* from the Message window.

testAvg

You should see the output of the *newAvg* function repeated four times.

Now, type this in the Message window:

```
newAvg (5)
```

What happens and why? What is the value of the second parameter within the *newAvg* handler? It defaults to `VOID`, as do all unspecified parameters, because only one argument was specified in the function call. What happens if we forget to specify the ***`fileName`*** when calling our *fileExists()* function created earlier?

Let's create a simple *divide* function (in reality you'd just use "/" to divide):

```
on divide a, b
  return float(a)/ b
end

put divide (5,5)
-- 1.0000
```

What happens if we forget to specify the parameter used as the divisor?

```
put divide (7)   -- This causes an error
```

Special Treatment of the First Argument Passed

We saw earlier that an error occurs when a local variable is used before it is assigned a value. Enter the code shown in Example 1-34 in a movie script, and recompile the script.

Example 1-34: Special Treatment of First Argument to a Function Call

```
on testFirstArg
  dummyHandler(x)
end testFirstArg
```

Isn't ***`x`*** an unassigned local variable? Shouldn't it generate a "*Variable used before assigned a value*" error message? Before answering that, let's alter *testFirstArg* and recompile the script:

```
on testFirstArg
  dummyHandler(x, y)
end testFirstArg
```

The variable ***`y`*** is also an unassigned local variable. Why does it generate an error message, if ***`x`*** did not? The answer lies in the unique way that Lingo treats the first argument to any function call.

Even though it is an error, Lingo does not complain if the first argument to a function call is an undeclared local variable (in this case ***`x`***).

Multiple scripts may contain handlers of the same name. When you call a function, Lingo must decide which script to look in first. If the first argument to the function call is a special entity called a *script instance* (see Chapter 2), Director runs the handler in that particular script rather than performing its usual search to

find the right script automatically. Lingo allows *anything* as the first argument to a function call because it does not verify script instances or handler names during compilation. (At runtime it will most likely cause an error, though).

If the first argument passed to a custom function call *is* a script instance, Lingo searches *only* that script instance for the specified handler. Therefore, you can not pass a script instance as the first parameter to a handler in a movie script because Lingo won't look in the movie script!

Suppose you want to call a movie script's handler from a Behavior, and suppose you want to pass in the Behavior's *script instance* as an argument. Assume that this is the Behavior script.

Example 1-35: Passing a Script Instance as an Argument

```
on mouseUp me
  displayInfo (me)
end
```

This is the movie script:

```
on displayInfo someScriptIntanceOrObject
  put the spriteNum of someScriptIntanceOrObject
end
```

What will happen? Director will issue a "*Handler not defined*" error because it will look for the *displayInfo* handler *only* in the Behavior script (but it won't find it). You can move the *displayInfo* handler into the Behavior script, in which case it will be available only to instances of that Behavior, or you can rewrite the example as shown in the code that follows. Note that we add a dummy `VOID` argument as a placeholder for the first argument to the function call, which allows our script instance to become the *second* argument. Because the second argument is not afforded any special treatment, Lingo searches the usual hierarchy and finds the *displayInfo* handler in the movie script!

Rewrite the *displayInfo* function call as:

```
on mouseUp me
  displayInfo (VOID, me)
end
```

In the *displayInfo* handler declaration in the movie script we must add a dummy parameter to "catch" the first dummy argument:

```
on displayInfo dummyParam, someScriptIntanceOrObject
  -- Use a dummy argument as the first parameter
  -- to allow an object to be passed as second argument.
  put the spriteNum of someScriptIntanceOrObject
end
```

A script instance or *child object* (which can be thought of as the same thing) is often *intentionally* passed as the first argument to force Lingo to look in the

correct script for the correct handler or property variables. See the example of property variables in the earlier "*Variable Types*" section and Chapters 12 and 13.

Optional Arguments and Varying Argument Types

Lingo's built-in commands typically complain if you pass the wrong number or unexpected type of arguments, but some accept arguments of different data types and/or a variable number of arguments ("*variable*" is used here to mean "*varying*," not a Lingo *variable*). For example, the second argument to *setaProp()* is either a property name or a property value, depending on whether the first argument is a property list or linear list. Likewise, the *puppetSound* command accepts either one or two arguments, and some arguments to the *puppetTransition* command are optional.

You can also design your custom handlers to allow a variable number of parameters or arguments of varying data types. This makes it easier to create generalized functions rather than multiple, highly similar versions. If you are creating a library of functions for yourself or others, it also makes those functions more flexible and easier to use by the calling routine.

Your function will typically require one or more mandatory arguments that should be placed at the beginning of the parameter list. Optional arguments should be placed after the required parameters. If the caller specifies fewer arguments than the number of parameters you are expecting, later parameters will be `VOID`.

The *playSound* example that follows accepts the name or number of a sound to play and an optional sound channel number. If no channel is specified, it plays the sound in channel 1. Note the use of *voidP()* to check whether the caller has specified the requested parameters.

Example 1-36: Function Accepting Varying Arguments

```
on playSound soundID, chan
  if voidP(soundID) then
    alert "A sound must be specified"
    exit
  end if
  -- Use channel 1 if the caller does not specify a channel
  -- Or specifies an invalid channel
  if voidP(chan) then
    set chan = 1
  else if not integerP(chan) or ¬
    (integer (chan) < 1  or integer (chan) > 8) then
    put "Channel should be an integer from 1 to 8"
    set chan = 1
  end if
  -- Play the sound
  puppetSound chan, the number of member soundID
  updateStage
end

-- This plays "woof" in channel 1 (the default)
playSound ("woof")
```

Example 1-36: Function Accepting Varying Arguments (continued)

```
-- This plays "bark" in channel 3
playSound ("bark", 3)
```

Note that we also check whether the channel number passed in is an integer and whether it is a valid sound channel number. Our example also accepts either a sound cast member's number or its name, just like as built-in *puppetSound* command. We could enhance the error checking to make sure the specified cast member is a sound.

Refer to Example 5-3 in Chapter 5, *Coordinates, Alignment and Registration Points*, in *Director in a Nutshell*. It accepts parameters in numerous formats and includes substantial error checking.

You can extend the *playSound* example to create a handler that accepts additional optional arguments, but note that the caller cannot pass a value for an optional argument unless all preceding arguments have been specified. For example, if we wanted to add a flag to *playSound* indicating whether to wait for the sound to play, we could add another optional parameter called *waitFlag*.

Example 1-37: Placeholder Arguments

```
on playSound soundID, chan, waitFlag
  -- Beginning of handler is the same code as example above
  -- but is omitted here for brevity
  puppetSound chan, the number of member soundID
  updateStage
  -- This will wait if waitFlag is non-zero. It will
  -- not wait if waitFlag is omitted, and therefore VOID
  if waitFlag then
   repeat while soundBusy (chan)
     nothing
   end repeat
  end if
end
```

Arguments and parameters are always matched up by *position*. The first argument in the function is assigned to the first parameter in the handler definition, and so on.

In this example, if the caller specifies only two arguments, the second argument will be used as the second parameter (*chan*). If the caller wants to specify *waitFlag* (the third parameter), he or she must specify three arguments, including a placeholder for *chan*.

```
-- The second argument is assumed to be the second
-- parameter and is mistakenly
-- interpreted as a channel number
playSound ("bark", TRUE)
-- Instead, use 1 as a placeholder for the chan parameter
playSound ("bark", 1, TRUE)
```

Variable-Length Parameter Lists

In the previous example, some arguments are optional, but the maximum number of arguments is known. You can also create handlers that accept an unknown (ostensibly unlimited) number of arguments. The *paramCount* property and *param()* function decipher an unknown number of arguments passed into a handler. *The paramCount* indicates the total number of parameters received and *param(n)* returns the n^{th} parameter.

Example 1-38: Variable Number of Parameters

```
on countParams
  put "Total Params:" && the paramCount
  repeat with n = 1 to the paramCount
    -- This statement prints out each parameter's
  -- number and its value by building a fancy string
    put "Param" && n & ":" && param(n)
  end repeat
end countParams

countParams ("Hello", "there", 5)
-- "Total Params: 3"
-- "Param 1: Hello"
-- "Param 2: there"
-- "Param 3: 5"
```

Note that no parameters are declared in the handler definition of *on countParams*. It will accept any number of parameters, as would be appropriate if we wanted to, say, average any number of values. If we expected a fixed number of parameters, we could instead declare some parameters (in this case *a*, *b*, and *c*) when we define our handler, such as:

```
on newCountParams a, b, c
  put "Total Params:" && the paramCount
  put "Param a:" && a
  put "Param b:" && b
  put "Param c:" && c
  put "Param 1:" && param(1)
  put "Param 2:" && param(2)
  put "Param 3:" && param(3)
end newCountParams

newCountParams ("Hello", "there", 5)
-- "Total Params: 3"
-- "Param a: Hello"
-- "Param b: there"
-- "Param c: 5"
-- "Param 1: Hello"
-- "Param 2: there"
-- "Param 3: 5"
```

We can access the first parameter as either *a* or *param(1)*. Likewise, we can access the second parameter as either *b* or *param(2)*, and so on. That is, *param(1)* is always the first parameter, not merely the first *unnamed* parameter.

Note that named parameters are easier to work with when you know how many to expect, but *param()* and *the paramCount* are more flexible. Use any combination of the two. Refer to Example 8-6 and 8-7 which use *the paramCount* and *param()* to take the sum or average of an indeterminate number of arguments.

Parameter Error Checking

The *playSound* example discussed previously ignores extraneous arguments (that is, if more arguments are specified than the number of parameters expected), as do many Lingo commands. You can always check *the paramCount* to warn the caller if too many or too few arguments are specified, such as:

```
if the paramCount > 3 then alert "No more than 3 please"
```

or

```
if the paramCount <> 4 then alert "Expected 4 params"
```

You can also check the type of each argument, as described in detail in Chapter 5:

```
if not integerP(param(1)) then
  alert "First parameter must be an integer"
  exit
    end if
```

The *verifyParams()* function shown in Example 1-39 checks whether the parameter(s) passed into a handler are of the expected data type(s). The details are fairly complex, but you don't need to understand them at this point.

You can often use a handler as a "*black box*." You don't need to know what happens inside the box; you need to know only what *inputs* it requires and what *outputs* it provides.

Likewise, you may provide handlers to others without supplying details on how they work. You need not understand all the magic as long as you trust the wizard behind the curtain. This book and its companion, *Director in a Nutshell*, try to dispel some of the mystery about how Director and Lingo work.

The *verifyParams()* function shown in Example 1-39 accepts a property list containing parameters and their expected data types (it checks only for integers, floats, and strings). See Chapter 6 if you don't understand lists, or just skip the details for now. *VerifyParams()* returns `TRUE` if the number and type of parameters are correct and `FALSE` otherwise. You can extend *verifyParams()* to handle more data types or to post an alert dialog instead of printing errors to the Message window.

Example 1-39: Verifying Parameters

```
on verifyParams  verifyList, numInput
  -- Check the number of parameters vs. the number expected
  set numExpected = count (verifyList)
  if numInput < numExpected then
    put "Too few parameters. Expected" && numExpected
```

Example 1-39: Verifying Parameters (continued)

```
    return FALSE
  else if numInput > numExpected then
    put "Too many parameters. Expected" && numExpected
    return FALSE
  end if
 -- Check each item in the list and its data type
  repeat with x = 1 to count (verifyList)
    set nextItem = getAt (verifyList, x)
    case (getPropAt(verifyList, x)) of
      #integer:
        if not integerP (nextItem) then
          put "Expected integer for parameter" && x
          return FALSE
        end if
      #float:
        if not floatP (nextItem) then
          put "Expected float for parameter" && x
          return FALSE
        end if
      #string:
        if not stringP (nextItem) then
          put "Expected string for parameter" && x
          return FALSE
        end if
      otherwise:
          put "Unsupported type for parameter" && x
          return FALSE
    end case
  end repeat
  return TRUE
end verifyParams
```

You can use *verifyParams()* to check if your routine is called with the correct number and type of arguments. This is useful for debugging your own code or for trapping errors if you distribute your code for others to use. *VerifyParams()* expects a property list containing each parameter and its expected data type. The following verifies whether *a* is an integer, *b* is a string, and *c* is a float. It also checks whether exactly three parameters have been received.

```
on myHandler a, b, c
  -- Make sure that we received the expect parameters
  if not (verifyParams([#integer:a, #string:b, #float:c],[LC]
       the paramCount)) then
    alert "Something was wrong"
    exit
  end if
  -- Otherwise everything is okay and we can proceed.
statements
end myHandler
```

Test it from the Message window:

```
myHandler (12.5, "a", 5.7)
-- "Expected integer for parameter 1"
```

```
myHandler (12, 6, 5.7)
-- "Expected string for parameter 2"

myHandler (5, "a")
-- "Too few parameters. Expected 3"
```

Reader Exercise: Modify *Verify Params()* to return an error string.

See also Example 8-4, "*Clipping a Value to a Valid Range*," in Chapter 8.

Congratulations!

Whew! You now have a foundation on which to build a greater understanding of Lingo. Even the most complex programs are built with simple components—variables, handlers, keywords, *repeat* loops, and *if* statements—so don't be intimidated. With patience, you can (de)construct very complicated programs. Refer to Table 18-1, for a list of all Lingo keywords so that you can distinguish them from variables and custom handler names. Look at examples of other people's Lingo code. Try to recognize the various pieces of the Lingo puzzle. (Remember diagramming sentences in English class, where you picked out the verbs, subjects, adjectives, and prepositional phrases?) Which items are variables? Which are keywords? Which are parameters? Which items are arbitrarily chosen by the programmer, and which are dictated by Lingo's grammar or syntax?

This single chapter has covered material from both beginner and intermediate programming courses that might be spread out over many months. We also touched on some very advanced concepts that will serve you well as you read the rest of this book. Don't be discouraged if you didn't understand a lot of it, or if you skipped the more intimidating parts. Re-visit this chapter frequently, and you'll find new treasures each time. It may seem hard to believe now, but when you look back on this chapter a year from now, most of the things that confused you will seem quite simple.

Most of this chapter applies to other programming languages you may encounter. If Lingo is your first programming language, rest assured that picking up additional languages becomes much easier. In Chapter 4 we compare Lingo to C/C++ so that you can see both the nitty-gritty details of Lingo and how other languages may differ.

Even though this "book work" may seem tedious, it will allow you to breathe new life into all your Director projects once you are out in the field. (Don't forget to save your test Director movie periodically).

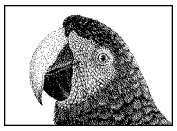
I leave this chapter with a reminder that I can point you in the right direction and even provide a map of the terrain and a steady compass, but you are ultimately your own navigator for the journey that lies ahead.

This exchange took place in the Director support forum:

Q: *What is* `TRUE`*?*

A: How about trying, *put TRUE*?

Where else but Lingo can you find out what is `TRUE` with a mere nine keystrokes?



CHAPTER 6

Lists

List Basics

There are two major types of Lingo lists—*linear lists* and *property lists*. Refer to Chapter 5, *Coordinates, Alignment, and Registration Points*, in *Director in a Nutshell* for an additional details on *rects* and *points*, which are list-style structures sharing characteristics of both linear and property lists.

Lists (called *arrays* in most languages) are convenient for storing and retrieving related data. A list consists of zero or more elements, which can be of any data type, enclosed in square brackets. Elements are separated by commas.

Simplify global variable management by using a single global list with multiple properties instead of individual global variables.

You do *not* need to allocate a specific amount of memory or number of elements for a list. Director handles the housekeeping as elements are added or deleted. Elements are often referred to by their *index* (that is, their position in the list).

The index of the first element's in a list is 1, not 0 as in some other languages. Do not use 0 or negative numbers as indices.

Linear Lists

A linear list is a simple comma-delimited list of elements, defined using the *list()* function or by enclosing items within square brackets. Each element can be of any

data type. Use *list()* without any elements, or empty brackets, [ ], to create a zero-element linear list. Here are some example linear lists:

```
set emptyList = [ ]
set myList1 = [1, 2, 3]
set myList2 = list (1, 7, #fumble, "Apples", 6.5, "Money")
```

Don't include brackets within the *list()* function unless you intend to create a list *within* a list.

```
put list([1,2,3])
-- [[1, 2, 3]]
```

Property Lists

Each element in a property list consists of a *property name:property value* pair (or simply a *property:value* pair), separated by a colon, such as `#myProp:7`. The value can be of any data type. The property name is usually a Lingo-style symbol (see "*Symbols*" in Chapter 5, *Data Types and Expressions*) but can be of any data type.

In fact, any alphanumeric characters used as a property name (except quoted strings) are converted to symbols.

Avoid using a float value as the property name because Director may not properly retrieve such elements, especially under Windows. Define a property list by enclosing one or more *property:value* pairs, separated by commas, within square brackets. Use a single colon within two brackets, [:], to create a zero-element property list. If your keyboard lacks square brackets, see Example 6-1. Here are some example property lists:

```
set emptyPropList = [:]
set myPropList1 = [#a:1, #b:2, #c:3]
set myPropList2 = [#name:"apples", #quantity:5]
```

Note that the property names ***fruit*** and ***calories*** are assumed to be symbols, not variables.

```
set myPropList3 = [fruit:"bananas", calories:12]
put myPropList3
-- [#fruit: "bananas", #calories: 12]
```

Property:value pairs are always added to or deleted from property lists as a unit. The *deleteAll()* command leaves a property list equal to [:] (empty, but still a property list).

Rects and Points

Rects and points are covered in detail in Chapter 5 in *Director in a Nutshell*. Rects and points are list-like structures that can be manipulated using many of Lingo's list functions. With some functions, rects and points behave like property lists; with other functions, they behave like linear lists. Rects and points are not affected usefully by commands that add or delete elements, and they should be used only with commands that get and set the value of existing elements. If you insist on playing Dr. Frankenstein, you can use *deleteAt()* to create a *franken*-rect with only

three elements, but *count()* will always return 4 for rects and 2 for points. See the "*Rect and Point Operations*" section later in this chapter.

Creating Lists

Table 6-1 shows the commands for defining lists.

A "vanilla" list contains *either* linear elements or *property:value* pairs, not both. A property list, however, can be a sub-element within a linear list, and vice versa.

Rects and points are defined with parentheses, not square brackets. Linear lists and property lists can both contain rects and points. A rect can consist of two points.

If your keyboard lacks square brackets, you can still create *linear* lists with the *list()* function. See Example 6-1 to create *property* lists.

The following utility creates an empty property list that can be populated with elements using other list commands.

Example 6-1: Creating Property Lists on Keyboards Without Square Brackets

```
on createPropList
  -- Left bracket is ASCII 91. Right bracket is ASCII 93
  return value(numToChar (91) & ":" & numToChar (93))
end createPropList
```

First, create the list and then add elements using *addProp*:

```
set myPropList = createPropList()
addProp (myPropList, #propA, 5)
put myPropList
-- [#propA: 5]
```

See also Example 6-27, "*Creating a Property List from Two Linear Lists.*"

Table 6-1: Commands to Define Lists

Command	Usage	Creates	Linear Lists	Property Lists	Points/ Rects
[ ]	Defines a linear list	Linear list	Yes	N/A	N/A
list(*elements*)	Defines a linear list	Linear list	Yes	Error	N/A
[:]	Defines a property list	Property list	N/A	Yes	N/A
point(*x*, *y*)	Defines a point	Point	N/A	N/A	Point
rect(*l*, *t*, *r*, *b*)	Defines a rect	Rect	N/A	N/A	Rect
duplicate(*list*)	Creates independent copy of a list, *sorted* if original was sorted	Same type as original list	Yes	Yes	Yes
value(string(*list*))	Creates independent *unsorted* copy of a list[1]	Same type as original list	Yes	Yes	Yes

[1] The list won't be considered "sorted" by Lingo, but the *value(string())* function does not change the order of the existing elements. *See* "*Sorting, Adding, and Changing List Elements*" later in this chapter.

Invalid List Declarations

The following are all *invalid* uses of lists.

The following causes a "Handler not defined" error because *myList* is not a property list, as required by *addProp()*:

```
set myList = [1,2,3]
addProp myList, #a, 1
```

The following causes an "Operator expected" error because *list()* can't create property lists:

```
put list (#a:5, #b:4)
```

The following causes a *"Property list did not start with a property name"* error because the third element includes a property name, but the first two did not:

```
set myList = [1,2,#a:4]
```

The following causes a "*Property or value missing*" error because the last element includes a property name but no property value:

```
set myList = [#lowell:1, #paul:2, #kenny:3, #sam]
```

The following causes an "*Operator expected*" error, even though the last element includes a colon, because it still lacks a property value:

```
set myList = [#lowell:1, #paul:2, #kenny:3, #sam:]
```

Valid List Declarations

The following are all acceptable ways of initializing lists.

Linear list with various data types:

```
set myList = [#a, void, 7.5, 2, "foo"]
```

Linear list containing property list as the second element:

```
set myList = [1, [#a:1, #b:5, #c:3], 6, "foo"]
```

Property list containing property lists and linear lists as values:

```
set myList =  [#someList: [#a:1, #b:5], #otherList: [1,2,4]]
```

Property list containing a rect and a point as values:

```
set myList =  [#myRect: rect(2,4,8,7), #myPoint: point(1,3)]
```

If you specify a value but omit the property name for a property list element, Lingo inserts a property name equal to the element's index. The following is not recommended, but it will work, as long as the value missing a property name is not a symbol.

```
set myList = [#a:1, "asd", #c:3, 5, 7.5, void]
put myList
-- [#a: 1, 2: "asd", #c: 3, 4: 5, 5: 7.5000, 6: Void]
```

If you add a variable to a list, the variable's *current* value is added to the list. If the variable's value changes, the list remains unchanged.

```
set x = 5
set myList = [x]
```

```
set x = 7
put x
-- 7
put myList
-- [5]
```

If you use a non-list variable as a value in a list, the list will include the variable's current contents. Even if the variable changes, the list will not. If you try to use a variable name as a property name it will be converted to a symbol:

```
set myVar = #someSymbol
set myList = [myVar:1]
put myList
-- [#myVar: 1]
```

Use the following to convert the symbolic property name (*#myVar*) in the previous example back to the value of the variable *myVar*:

```
put value(string(getPropAt (myList, 1)))
-- #someSymbol
```

This technique can be used to create a list of variable names, rather than a list of the values of the variables. In most cases, you are better off simply using a list rather than trying to track variables within a list.

Assigning, Passing, and Duplicating Lists

Lists can be assigned to any local, global, or property variable. (A property variable (which is defined using the *property* keyword) should not be confused with a Lingo property (such as *the clickLoc*), a property list, or a property *of* a list. Refer to "*Child Object Properties*" later in this chapter and to Chapter 1, *How Lingo Thinks*, and Chapter 12, *Behaviors and Parent Scripts*.)

A list variable *points* to the *beginning* of the list in memory; it does *not* contain a copy of the entire list. Lists are passed by *reference*, not by value, allowing for efficient passing of large lists between handlers.

If you assign one list variable to another variable, or pass a list variable as an argument to a handler, both entities point to the same list. If either list changes, both change because they point to the same data in memory.

In Example 6-2, deleting the elements in *listA* removes them from *listB*.

Example 6-2: Two Variables Pointing to a Single List

```
set listA = [1,2,3]
set listB = listA
put listB
-- [1, 2, 3]
deleteAll (listA)
put listB
-- []
```

(The *deleteAll* command used here has worked since Director 4, but it was undocumented until Director 6.)

Use either the *duplicate(`list`)* or *value(string(`list`))* function to make an independent copy of a list. (Using *value()* alone has no effect, and the lists remain linked.)

In Example 6-3, note that changes to *`listA`* no longer affect *`listB`*.

Example 6-3: Dissociating Two Lists

```
set listA = [1,2,3]
set listB = duplicate (listA)
deleteAll (listA)
put listA
-- []
put listB
-- [1, 2, 3]
```

If you pass a list into a handler and modify that list within the handler, the original list in the calling routine is modified as well.

Example 6-4: Modifying Lists Passed to a Handler

```
on listPass
  set myList = [3,2,1]
  put "MyList started as" && myList
  adjustList (myList)
  put "MyList ended up as" && myList
end listPass

on adjustList someList
  sort someList
end adjustList
```

Note that *`myList`* is altered by *adjustList():*

```
listPass
-- "MyList started as [3, 2, 1]"
-- "MyList ended up as [1, 2, 3]"
```


Don't modify the original list passed into a handler unless that is the specific intent, as with the *deleteRange()* handler in Example 6-15. Instead, work on a *copy* of the list, and return the altered version via a *return* statement.

In Example 6-5, I've modified *adjustList()* to create a separate list that it returns to the caller, and I've also modified *listPass()* to receive the returned list.

Example 6-5: Using a List in a Handler Without Modifying the Original List

```
on listPass2
  set myList = [3,2,1]
```

Example 6-5: Using a List in a Handler Without Modifying the Original List

```
  put "MyList started as" && myList
  set workList = adjustList2 (myList)
  put "MyList ended up as" && myList
  put "WorkList ended up as" && workList
end listPass2

on adjustList2 someList
  set workList = duplicate(someList)
  sort workList
  return workList
end adjustList2
```

Note that *myList* is *not* affected by the changes made within *adjustList2()*.

```
listPass2
-- "MyList started as [3, 2, 1]"
-- "MyList ended up as [3, 2, 1]"
-- "WorkList ended up as [1, 2, 3]"
```

Using Lists to Return Multiple Values from a Handler

This section assumes that you have read and understood *"Parameters and Arguments"* in Chapter 1, *How Lingo Thinks*. Arguments of most data types are passed to handlers by *value*. If you want to modify a single variable, it is easy to set it to the return value of a function, such as:

```
set  x = power (10, 2)
```

But, suppose you wanted to swap the value of two integer variables. You'd need to modify them *both*, but a function can return only *one* value. This won't work:

```
set x = 1
set y = 2
swapInts (x, y)
```

You can *not* write a *swapInts()* function that modifies two variables because it is the *value* of the variables, and not the variables themselves, that are passed to *swapInts()*. In C, you could manually pass the *address* of the variables (that is, pass them *by reference*) if you wanted to swap two variables. In Lingo you have to use a third variable to swap them manually (and you can not use a function to do it), such as:

```
set temp = x
set x = y
set y = temp
```

Arguments that are *lists* or *objects*, however, are passed by reference. The called function receives the list itself, not just a copy of its contents. Therefore, you can pass in, and return, a series of arguments as elements within a list.

Example 6-6: Using Lists to Return Multiple Values from a Handler

```
on mouseUp
  set myList = [#low: the mouseH, #high: the left of sprite 5]
  if the low of myList > the high of myList then
    swapHighLow (myList)
```

Example 6-6: Using Lists to Return Multiple Values from a Handler (continued)

```
  end if
  put "myList has been swapped:" && myList
end

on swapHighLow inList
  set temp = the low of inList
  set the low  of inList = the high of inList
  set the high of inList = temp
end
```

Example 6-6 is merely for illustration. In practice you would simply sort the list.

Cleaning Up After Lists

Director deallocates the memory used for a list when no variables still point to it. To deallocate a list and free memory, assign all variables that pointed to the list to 0.

```
set myList = 0
```

Clear all global lists (and other objects) in MIAWs to allow them to be removed from memory.

Multidimensional Arrays

Linear lists are one-dimensional. Property lists are essentially lists of 2 by *n* elements. You can create multidimensional arrays using lists *within* lists. Irv Kalb discusses his multidimensional array object in the *Lingo User's Journal* (*September 1995 Volume 1, Number 3*) (*http://www.penworks.com*).

This linear list of three property lists might represent a high score chart or student test scores.

Example 6-7: Creating and Reading Multidimensional Lists

```
on printHighScores
   set highScores = [ ¬
     [#name: "Jane", #score: 85, #level: 8], ¬
     [#name: "Dick", #score: 75, #level: 7], ¬
     [#name: "Spot", #score: 95, #level: 9]]
  repeat with x in highScores
    put the name of x && "scored" && the score of x
  end repeat
end

printHighScores
-- "Jane scored 85"
-- "Dick scored 75"
-- "Spot scored 95"
```

In the previous example, you could instead use a double repeat loop, extracting the data using indices instead of property names. Note the use of the nested *getAt()* statements in the following example:

```
repeat with i = 1 to count (highScores)
  repeat with j = 1 to count (getAt(highScores, i))
    put getAt(getAt (highScores, i), j)
  end repeat
end repeat
```

C programmers will notice that Lingo's syntax for accessing list elements is horribly verbose.

Lists as a Database

Lists operations are much faster than text string manipulation (see Chapter 7, *Strings*). Sort your lists for faster access. Lists are capable of maintaining a database of several thousand elements. You can use lists to store and manipulate data at runtime, and you can use the FileIO Xtra to read the data from or write the data to an external text file for permanent storage. Refer to Chapter 14, *External Files*, for details.

For more extensive database capabilities, consider a third-party Xtra, such as those listed in 10, Using Xtras, in Director in a Nutshell (also available at http://www.zeusprod.com/nutshell/links.html)

Keep in mind that Director's Cast is a multimedia database. You can use lists or database Xtras to store castmember names but use the Cast itself to store the media. For example, you could create a huge database of songs, cross-referenced by artist, title, and genre, and then play them from the cast using *puppetSound* (or from external files using *sound playFile*).

Lingo List Commands

The names of Lingo's list-related commands suck. You'll probably leave the *O'Reilly* bookmark (included at the back of this book) planted firmly in this chapter. See "*Making Sense of List Commands*" later in this chapter for hints on finding the right list command, or create *wrapper scripts* that use names that are easier to remember but simply call a single Lingo function, such as:

```
on getPropByValue myList, value
  -- Return the value obtained from getOne
  return getOne (myList, value)
end
```

Refer to Chapter 3, *Lingo Coding and Debugging Tips*, for details on wrapper scripts.

Some list commands work with both linear lists and property lists, but others work exclusively with one or the other. The same list command may behave differently or accept different arguments when used with different list types. There are multiple commands that have the same effect on certain list types.

Using the wrong type of list, or a non-list, as the first argument to a list function will cause the (highly misleading) "*Handler not defined*" error. See Examples 3-21c through 3-21f.

Use *listP()* or *ilk()* to check if your variable is the correct type of list. (See Example 5-3, "*Avoiding Evaluation of the Second Clause in a Compound Expression*," under "*Boolean Properties and Logical Operators*" in Chapter 5 for important details.) If necessary, initialize the list inside the current handler, or declare it as a global variable if it is defined elsewhere.

Many list functions do not return a meaningful value. Instead, they modify the list used as the first argument. You cannot use the following syntax because *add()* does not return a list:

```
set myList = add([], 7)
```

Use this instead:

```
set myList = []
add (myList, 7)
```

Making Sense of List Commands

List commands often use one attribute of an element—its position (*index*), property name (*property*), or property value (*value*)—to determine some other attribute. For example, you can read the property at a certain position or the value associated with a certain property.

These tips may help you make sense of the list commands:

- The first argument to any list command must *always* be a valid list. (If not, you'll be greeted with a *"Handler not defined"* error.)
- If a value is required, it is *always* the *last* argument in the parameter list (that is, specified after the position or property name).
- The "At" commands—*addAt()*, *setAt()*, *getAt()*, *deleteAt()*—all use an element's *position* in the list as the second argument.
- Specifying an index less than 1 or greater than the number of elements in the list will cause an *"Index out of range"* error with most, but not all, functions. *AddAt()* and *setAt()* accept indices beyond the end of the current list, adding elements as needed. Check the *count()* of a list to ensure that your index is in range. Some property list functions fail if you specify a nonexistent property name, but *setaProp()* will append the specified property, if necessary.
- Most "Prop" commands—*addProp()*, *setProp()*, *getProp()*, *getPropAt()*—work with property lists but not linear lists.
- The "aProp" commands—*getaProp()*, *setaProp()*, *deleteProp()* (there is no *deleteaProp*)—are intended for property lists but emulate *getAt()*, *setAt()*, and *deleteAt()* when used with linear lists.

- Only the commands that *create* lists (shown in Table 6-1) *return* a list as the result of the function call. Commands that *modify* lists do not return a new list, but rather modify the list passed as the first argument "in place."
- The "Get" and "Find" commands—*getProp()*, *getAt()*, *getLast()*, *getOne()*, *getPos()*, *getProp()*, *getPropAt()*, *findPos()*, *findPosNear()*—always return a *single* datum, such as a position, property, or value. You must use two separate commands to retrieve, say, both an element's property and its value.
- Some commands return error codes, whereas others issue alert dialogs if the parameters are invalid.

Sorting, Adding, and Changing List Elements

Lingo can optionally sort a list using the *sort* command. Sorting affects the order in which subsequent elements are added, and it affects the speed of commands that access the list by value (but not by position). The *add*, *findPos()*, *findPosNear()*, and *getOne()* commands work faster with sorted lists. (The *findPosNear()* function is intended for sorted lists and does not work fully with unsorted lists.) A list remains sorted unless some command cancels its "sort-edness". I refer to such a list as *unsorted* because Lingo no longer considers it truly sorted although existing elements remain ordered. A list's sort-edness affects the speed with which its elements accessed and how *future* elements are added.

Sort Order

A sort is always performed in *ascending* order.

Linear lists are sorted by *value*.

Example 6-8: Sort Order of Linear and Property Lists

```
set myList = [1, 7, #fumble, "Apples", 6.5, "Money"]
sort myList
put myList
-- [1, 6.5000, 7, "Apples", #fumble, "Money"]
```

Property lists are sorted by *property name*.

```
set myList = [#r:1, #q:7, #g:#fumble, #b:"Apples", ¬
     #a:6.5, #c:"Money"]
sort myList
put myList
-- [#a: 6.5000, #b: "Apples", #c: "Money", #g: #fumble, ¬
    #q: 7, #r: 1]
```

When sorting elements of different types, note that numbers are sorted before strings and symbols. Symbols are treated like strings and are sorted alphabetically, comingled with strings (although the string "a" will be placed before the symbol #a). Refer to Appendix C, *Case-Sensitivity, Sort Order, Diacritical Marks, and Space-Sensitivity*, for details on the exact sort order of strings with varied case, strings and symbols with diacritical marks, and non-alphanumeric characters (which differ on the Macintosh and Windows).

Recursive Sorting

If you sort a list that in turn contains *other* lists, *only* the primary list is sorted. Example 6-9 recursively sorts all sublists (to any number of levels) within a list using recursion.

Example 6-9: Recursively Sorting Sublists

```
on recursiveSort inputList
  -- Sort whatever list is passed in. This will be the
  -- top-level list at first, but will be a sublist if
  -- this is called recursively.
  sort inputList
  repeat with thisElement in inputList
    -- If this element is itself a list, keep sorting
    if listP(thisElement) then
      recursiveSort(thisElement)
    end if
  end repeat
end recursiveSort
```

Example 6-9 modifies the original list passed into the handler. It is left as an exercise to the reader to write a nondestructive version that returns a sorted listed without modifying the original list used as the input. (Bear in mind that you can't simply duplicate the list each time the routine is called because it is called recursively and you must work on the same list following the initial call.)

See Example 8-8, "*Recursively Counting Elements in a List*," in Chapter 8, *Math (and Gambling)* for another example of recursion.

Reverse Sorting and Reverse Ordering

For a descending sort, you can either access a sorted list backwards or create a list that is sorted in reverse order.

This *reverseSort()* handler uses the *reverseList()* handler defined in Example 6-11 to return an inverted, independent copy of the original sorted list. The reverse sort is a one-time operation; new elements will not be automatically sorted.

Example 6-10: Reverse Sort

```
on reverseSort inList
  if listP(inList) then
    set newList = duplicate (inList)
    sort newList
    return reverseList (newList)
  else
    alert "Did not receive a valid list"
    return VOID
  end if
end reverseSort
```

You can use this *reverseList()* handler to reverse unsorted lists as well.

Example 6-11: Reversing Unsorted Lists

```
on reverseList inList
  if not listP(inList) then
    alert "Did not receive a valid list"
    return VOID
  end if
  set listCount = count(inList)
  if ilk (inList) = #propList then
  --reverse a property list
    set newList = [:]
    repeat with x = listCount down to 1
      addProp newList, getPropAt (inList, x), [LC]
         getAt (inList, x)
    end repeat
  else
  --reverse a linear list
    set newList = [ ]
    repeat with x = listCount down to 1
      add newList, getAt (inList, x)
    end repeat
  end if
  return newList
end reverseList
```

Sorting and List Performance

Sorting a list takes negligible time unless the list is very long. Elements are *retrieved* more quickly from sorted lists, but *adding* elements to a sorted list is marginally slower. For short lists you won't notice any performance difference, but if you are accessing a large list many times, you should sort the list. If you are adding many elements, it is preferable to sort the list once after all (or most) elements are added.

Creating a list is much slower than accessing its elements, so you should initialize a list only once. Building a list incrementally is *much* slower than declaring an entire list at once. If the list is fixed, declare it using one of the functions in Table 6-1, rather than adding elements manually. Avoid repeatedly initializing a list while looping in a frame. Initialize it once in the previous frame, and access it via a global declaration instead.

Windows machines are generally faster than comparable Macintoshes for list creation and access, but it is unlikely you will notice the difference unless using very large lists.

Table 6-2 shows some speed comparisons for various operations. All operations were performed 10,000 times on a simple linear or property list containing 2750 elements. Note that *getAt()* operates at the same speed for sorted and unsorted lists because it accesses elements by their position only. *GetAt()* is marginally faster than *getaProp()*, even for sorted lists. These tests accentuate very minor

absolute differences, so you might not notice any speed differences for small lists or fewer operations.

Table 6-2: Sorted List versus Unsorted List Speed Comparison

CPU/Speed	getaProp() Unsorted	getaProp() Sorted	getAt() Sorted or Unsorted
Mac 68040 (33 MHz)	2390 ticks	99 ticks	74 ticks
PPC 601 (66 MHz)	1056 ticks	38 ticks	28 ticks
PPC 604e (225 MHz)	250 ticks	10 ticks	6 ticks
Pentium (166 MHz)	528 ticks	28 ticks	22 ticks

Adding Elements to Sorted and Unsorted Lists

Once a list is sorted, Lingo inserts new elements in sorted order, when using *add()*, *addProp()*, *setProp()*, and *setaProp()*. A list created by *duplicate(`list`)* is sorted if the original list was sorted. A list created using *value(string(`list`))* is not sorted by default.

The *append()* and *addAt()* commands cancel a list's "sort-edness," but existing elements remain in their previous order. Subsequent elements added with any command, including *add()*, are *not* inserted in sorted order unless the list is resorted.

Although undocumented, *add()* can be used with the same syntax as *addAt()*—namely *add(`list`, `position`, `value`)*—which unsorts the list, as would *addAt()*.

There is no *addPropAt()* command. Properties are added to the end of unsorted property lists or sorted by property name in sorted property lists.

When using *addAt()* or *add()*, if you specify a position beyond the end of the list, Lingo fills in the intervening items with zeroes.

Example 6-12 creates a 10-element list full of zeroes.

Example 6-12: Populating a List with Zeroes

```
set myList = [ ]
addAt(myList, 10, 0)
put myList
-- [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

List Commands That Add and Modify Elements

Table 6-3 shows commands that add, change, and sort elements. None of these commands return a meaningful value, but rather modify the list passed as the first argument. Only the *setAt()* and *set the `property` of `list`* commands work with

rects and points because the number and position of elements in a rect or point are fixed.

Table 6-3: Commands That Add, Change, and Sort Elements

Command	Usage	Linear Lists	Prop Lists	Points/ Rects
add(`list`, `val`)	Inserts value in sorted order or appends value[1]	Yes	Error[2]	Ignored
addAt(`list`, `index`, `val`) add(`list`, `index`, `val`)	Inserts value *before* element at specified position[3]	Yes	Error[2]	Ignored
addProp(`list`, `prop`, `val`)	Inserts *property:value* pair in sorted order or appends pair[1]	Error[2]	Yes	Error[2]
append(`list`, `val`)	Appends value to list[3]	Yes	Error[2]	Ignored
setaProp(`propList`, `prop`, `val`)[4] setaProp(`list`, `index`, `val`)[5]	Replaces or inserts[1] value by property name or position	Yes, by index[5]	Yes, by prop name[4]	Yes, by prop name[4]
setAt(`list`, `index`, `val`)[6]	Replaces item by position[3]	Yes	Yes	Yes
setProp(`list`, `prop`, `value`)	Replaces value by property name	Error[2]	Yes[7]	Error[8]
set the `property` of `list`[9]	Sets property's value by its property name	Error	Yes	Yes
sort(`list`)	Sorts linear list by value or property list by property	Yes	Yes	Ignored

[1] Inserts item at end of unsorted list or in sorted order for sorted lists.
[2] Causes "*Handler not defined*" error.
[3] Ignores the sort order and cancels sort-edness of list. Inserts item at specified location or appends to list.
[4] For property lists, the second argument to *setaProp* is assumed to be a property name, not an index, even if it is an integer. The property will be added if it does not exist, as with *addProp()*. For rects and points, the second argument to *setaProp* must be one of the valid rect or point properties, and not an integer (see Table 6-9). *SetaProp* also accepts a child object instance instead of a list as the first parameter.
[5] For linear lists, *setaProp* is identical to *setAt* and requires an integer index as the second argument. Specifying anything else causes an *"Integer Expected"* error.
[6] For linear lists, if the index is beyond the limit of the list, *setAt()* will append to the list. For property lists, the index must be within the limits of the list. Use the index 1 or 2 for points or an index from 1 to 4 for rects.
[7] For property lists, *setProp* causes a *"Handler not defined"* error if the property does not exist. Use *setaProp()* instead.
[8] *setProp* always fails for rects and points. Use *setaProp* instead.
[9] *Set the* `property` *of list* causes a "Property not found" error if the property does not exist. For property lists, use *setaProp()* instead. For rects and points, use only the valid rect and point properties.

Deleting Elements

Table 6-4 shows the commands that delete elements from a list. Deleting an element from a property list deletes its *property:value* pair. These commands should not be used with, and are generally ignored by, rects and points (using *deleteOne()* can corrupt a rect or point).

The *deleteOne()* function is case-sensitive when locating a string value to delete. It is *not* case-sensitive when accessing elements by symbol names instead of strings. Also see *getOne()*, *getPos()*, and Example 6-13.

Note the changing contents of `myList` in the examples below. In each succeeding example, `myList` contains the value from the previous command. *DeleteOne()* returns 1 if successful or 0 otherwise.

Example 6-13: Case-Sensitive Deletion of String Values

Define a list with strings of varied case.

```
set myList = [#a:"banana", #b:"BaNaNa"]
put myList
-- [#a: "banana", #b: "BaNaNa"]
```

Try to delete an item by matching the string "*BANANA*" in the list.

```
put deleteOne (myList,"BANANA")
-- 0
```

The list doesn't change because "*BANANA*" (all capital letters) doesn't match any items, so none were deleted.

```
put myList
-- [#a: "banana", #b: "BaNaNa"]
```

This deletes property #b (not property #a) because the match is case-sensitive:

```
put deleteOne (myList,"BaNaNa")
-- 1
put myList
-- [#a: "banana"]
```

This deletes property #a because the match is again case-sensitive:

```
put deleteOne (myList,"banana")
-- 1
put myList
-- [:]
```

Note that the final list is an empty *property* list, not an empty *linear* list.

Example 6-14 demonstrates that symbols are case-insensitive when used with list functions.

Example 6-14: Case-Insensitive Deletion of Symbolic Values

Define a list with symbols (instead of strings) of varied case.

```
set myList = [#a:#banana, #b:#BaNaNa]
```

Note that Director immediately converts symbols with the same name to the same case:

```
put myList
-- [#a: #banana, #b: #banana]
```

Below, property #a is deleted because it is the first property with a symbol value matching *#BaNaNa* (regardless of case).

```
put deleteOne (myList,#BANANA)
-- 1
put myList
-- [#b: #banana]
```

Likewise, any case-insensitive match is sufficient to delete property #b:

```
put deleteOne (myList,#BANana)
-- 1
put myList
-- [:]
```

Table 6-4: Commands That Delete List Elements

Command	Usage	Return Value	Linear Lists	Prop Lists
deleteAll(*list*)[1]	Deletes all elements from a list, leaving list type unchanged	0	Yes	Yes
deleteAt(*list*, *index*)	Deletes element by position	0	Yes[2]	Yes[2]
deleteOne(*list*, *val*)	Deletes first element with specified value	0: failure 1: success	Yes	Yes
deleteProp(*list*, *prop*)	Delete first element with specified property name	0: failure[3] 1: success	Yes[2,3]	Yes

[1] The *deleteAll* command is convenient for deleting all elements without knowing (or changing) a list's type. It has worked since Director 4 (although it was undocumented until Director 6). See also the custom *deleteRange()* handler in Example 6-15.
[2] Invalid indices cause an "*Index out of Range*" error. Check the range first.
[3] Using *deleteProp()* with a linear list requires an integer instead of a property, in which case it emulates *deleteAt()*; *deleteProp()* is intended for property lists and always returns 0 for linear lists.

You can check whether *deleteOne()* and *deleteProp()* returned `TRUE` (1) to determine whether the operation succeeded. Bear in mind that *deleteProp()* always returns 0 for linear lists.

When you delete an item in a list, all subsequent items move up one place (their indices change). Delete items in reverse order from the list to avoid any problems.

The *deleteAt()* command deletes only one element at a time and *deleteAll()* deletes them all. The *deleteRange()* handler below deletes a range of elements. It first ensures that *deleteFrom* is less than *deleteTo*, then it deletes elements from the

highest index down to the lowest. (Reader quiz: What happens if you delete from the lowest to highest index instead?)

Example 6-15: Deleting a Range of Elements

```
on deleteRange inList, deleteFrom, deleteTo
 -- Ensure that deleteFrom is less than deleteTo
  if deleteFrom > deleteTo then
    set temp = deleteFrom
    set deleteFrom = deleteTo
    set deleteTo = temp
  end if
  --  Prevents errors from invalid index ranges
  set deleteFrom = max (1, deleteFrom)
  set deleteTo = min (count (inList), deleteTo)
  -- Delete backwards in list
  repeat with x = deleteTo down to deleteFrom
      deleteAt (inList, x)
  end repeat
  return inList
end deleteRange

put deleteRange ([#a, #b, #c, #d, #e, #f], 2, 4)
-- [#a, #e, #f]
```

There is no built-in *deleteLast()* function. Use:

```
deleteAt (inList, count (inList))
```

Getting Info About, and Elements from, Lists

Table 6-5 shows commands that provide information about, or extract elements from, a list.

GetOne(), *getPos()*, and *findPos()* are *case-sensitive* when searching by strings values! They are *not* case-sensitive when accessing elements by symbol names instead of strings.

Note the case of each string and the result of each query in Example 6-16..

Example 6-16: Case-Sensitive Searching of String Values

```
set myList = ["banana", "Banana"]
put getPos (myList ,"banana")
-- 1
put getPos (myList ,"Banana")
-- 2
put getPos (myList ,"BANANA")
-- 0
```

Again, note the case of each string and the result of each query:

```
set myList = [#a:"banana", #b:"Banana"]
put getOne (myList,"banana")
-- #a
put getOne (myList,"Banana")
-- #b
put getOne (myList,"BANANA")
-- 0
```

As in Example 6-14, getOne(), getPos(), and findPos() are case-insensitive when searching for symbols.

Table 6-5: Commands That Read from Lists

Command	**Usage**	**Returns**	**Linear Lists**	**Prop Lists**	**Points/ Rects**
count(*list*)	Counts items in list	item count	Yes	Yes	Yes
ilk(*list*), listP(*list*), or ilk*(list, #type*)	Determines list's type	See Table 6-6	Yes	Yes	Yes
findPos(*list*, *prop*)[1]	Finds the position of the first occurrence of property	position, or VOID if not found	No	Yes	No
findPosNear (*list*, *prop*)[2]	Finds position (from 1 to *count()*+1) at which specified property name belongs in list	position at which property belongs	No	Yes	No
getaProp (*list*, *prop*)[3] getaProp(*list*, *index)*[4]	Gets value by property name (for propLists, rects, and points) or position (for linear lists)	value, or VOID if property not found[5]	Yes, by index[4,6]	Yes, by prop name[3]	Yes, by prop name[5]
getAt(*list*, *index)*	Gets value by position	value[6]	Yes	Yes	Yes
getLast(*list*)	Gets value of last element	value, or VOID if list is empty	Yes	Yes	Yes
getOne(*list*, *val*)[7]	Gets first property or position that matches value	prop, or position, or 0 if not found	Yes[7]	Yes[8]	Yes[7]
getPos(*list*, *val*)	Gets first position that matches value	position, or 0 if not found	Yes[7]	Yes	Yes[7]
getProp(*list*, *prop*)	Gets value of property	value	Error[9]	Yes[10]	Error[11]

Table 6-5: Commands That Read from Lists (continued)

Command	Usage	Returns	Linear Lists	Prop Lists	Points/ Rects
getPropAt(*list*, *index*)	Gets property name by position	property name	Error[9]	Yes[6].	Yes[6]
min(*list*), max(*list*)	Determines minimum or maximum value in list[12]	min or max value	Yes	Yes	Yes
the `property` of *list*	Gets the value of a property	value	Error	Yes[13]	Yes[13]

[1] *FindPos()* is intended for property lists and expects a property as an argument. It does not return useful information for linear lists, nor for rects and points, even when used with valid rect and point properties. Use *getaProp()* for rects and points instead.
[2] *FindPosNear()* is intended for *sorted* property lists. It returns the position of an exact match or the position at which the specified property would be inserted were it added to the list. The return value is between 1 and *count(list)*+1. For unsorted lists, it returns the position of an exact match; otherwise it returns *count(list)*+1. For non-property lists, it does not return meaningful information.
[3] For property lists, the second argument to *getaProp()* is assumed to be a property name, not an index, even if it is an integer. For rects and points, the second argument to *getaProp()* must be one of the valid rect or point properties (see Table 6-7). Use of an integer position causes a *"Symbol expected"* error.
[4] For linear lists, *getaProp()* is identical to *getAt()* and requires an integer index as the second argument. Specifying anything else causes an *"Integer Expected"* error.
[5] If the property is not found within a rect or a point, *getaProp()* causes a "Property not found" error. If the property is not found within a property list, *getaProp()* returns `VOID`, which is indistinguishable from a property with a value of `VOID`.
`VOID` may be returned as an error code:

```
put getaProp ([#a:1, #b:3, #c:void], #d)
-- Void
```

`VOID` may also be returned as a property's value:

```
put getaProp ([#a:1, #b:3, #c:void], #c)
-- Void
```

[6] Invalid indices cause an *"Index out of Range"* error. Check range first.
[7] For linear lists, rects, and points, *getOne()* and *getPos()* are identical and return the position at which the first occurrence of the value is found or 0 (zero) if it is not found.
[8] For property lists, *getOne()* returns the name of the first property with the specified value. If not found, it returns 0 (zero), which is indistinguishable from a property named 0, so you shouldn't use 0 as a property name.
Zero may be returned as an error code:

```
put getOne ([#a:1, #b:3], 7)
-- 0
```

Zero may also be returned as the matching property's name:

```
put getOne ([#a:1, 0:3], 3)
-- 0
```

[9] Causes a "*Handler not defined*" error.
[10] For property lists, *getProp()* causes a "*Handler not defined*" error if the property does not exist. Use *getaProp()* instead.
[11] *getProp()* always fails for rects and points. Use *getaProp()* instead.
[12] *min()* and *max()* work with all list element types, including strings. (They are case-sensitive; see Appendix C.)
[13] Using *the* `property` *of* `list` causes a *"Property not found"* error if the property does not exist. For property lists, use *getaProp()* instead. For rects and points, use only valid rect and point properties.

Performing Math with Lists

You can use standard math operators (+, -, *, /, and mod) to alter the contents of a single list as shown in Example 6-17.

Example 6-17: Math Operations Performed on Entire Lists

```
put [1,2,4] * -6
-- [-6, -12, -24]
put [1,2,4] - 7
-- [-6, -5, -3]
put -[1,2,4]
-- [-1, -2, -4]
put [1,2,4] / 2.0
-- [0.5000, 1.0000, 2.0000]
put [1,2,4] mod 2
-- [1, 0, 0]
```

You can also perform calculations using two lists as shown in Example 6-17.

The resulting list is the length of the shorter of the two lists.

Example 6-18: Math Operations Performed with Two Lists

```
put [1,2,4] + [7,8,9,10]
-- [8, 10, 13]
put [1,2,4] - [7,9]
-- [-6, -7]
put [1,2,4,12,15] * [4,5,6,12]
-- [4, 10, 24, 144]
put [10,20,40] / [4,5,6]
-- [2, 4, 6]
put [10,20,40] mod [4,5,6]
-- [2, 0, 4]
```

See also Example 8-6, "*Summing a List of Values*" and other examples in Chapter 8.

Lists in Expressions

Lingo treats lists within expressions in a somewhat arbitrary, if not capricious, manner. Testing is the only way to determine how Lingo will interpret a given expression, usually falling into one of these categories:

The list is treated as a single entity.

Note that the entire list is converted to a string:

```
put string (myList)
-- "[1, 3, 4]"
```

The logical expression is not applied to each element. The list itself is a non-zero entity, so the logical *not* of the list is `FALSE`.

```
put not ([0,2,3])
-- 0
```

The entire list, not just the first element, is compared to the other operand when using = or <>, such as:

```
put ([4,5,6]) <> 4
-- 1
put ([4,5,6]) = 4
-- 0
```

Contrast this with the inequality operators below.

Lists are compared on an element-by-element basis. Two variables pointing to two different lists are considered equal if the lists contain identical elements.

```
set x = [1, 2, 3]
set y = [1, 2, 3]
put x = y
-- 1
```

The first element of the list is used in the expression.

In the following example, the comparison is made between the first element of the list and the right side of the expression:

```
put ([4,5,6]) > 3
-- 1
put ([4,5,6]) > 5
-- 0
```

Contrast this behavior with that of = and <>. Comparisons involving two items, with <, >, and =, can *all* be `FALSE` because of Lingo's differing rules for evaluating inequalities and equalities.

The operation is invalid or otherwise ignored.

Attempting to use a list in a Boolean expression causes an "*Integer expected*" error:

```
if (myList) then put "foo"
```

Both of these comparisons are `FALSE`, so nothing is printed:

```
if myList = FALSE then put "It's FALSE"
if myList = TRUE then put "It's TRUE"
```

To check the existence of a list, use *listP()*. One of the following two statements will cause a message to print:

```
if listP(myList) = FALSE then put "It's FALSE"
if listP(myList) = TRUE then put "It's TRUE"
```

The *float()* function is ignored when used with lists:

```
put float ([1,2,3])
-- [1, 2, 3]
```

A list is a complex data structure that cannot be integer-ized:

```
put integer ([9,8,7])
-- Void
```

The results returned from various Lingo functions are not always consistent. Note that a list is an object according to *objectP()* but not according to *ilk()*:

```
put objectP([1,2,3])
-- 1
```

```
put ilk([1,2,3], #object)
-- 0
```

Non-list Variables in Compound Expressions

In the following expression, if *myList* is not a list, the expression *count(myList)* will cause a "*Handler not defined*" error (you can't use *count()* on something that is not a list).

```
if listP(myList) and count(myList) > 5 then
  -- whatever
end if
```

Use the construct shown in Example 6-19 instead:

Example 6-19: Preventing Non-list Arguments from Being Passed to List Commands

```
if listP(myList) then
  if count(myList) > 5 then
    -- whatever
  end if
end if
```

See Example 5-3, "*Avoiding Evaluation of the Second Clause in a Compound Expression*," under "*Boolean Properties and Logical Operators*" in Chapter 5 for details on logical expression evaluation.

Commands by List Type

The following section details the commands that perform common operations on each type of list. Refer to prior tables for functions common to all lists, such as *min()*, *max()*, and *count()*.

Determining a List's Type

You may need to determine a list's type to decide which commands to use with it. You can use *listP()* to determine whether a datum is any of the four possible types of lists, and you can use *ilk()* for more detailed information about a list's type. You can use *listP()* in *if...then* statements and *ilk()* in *case* statements (see Chapter 5 for a comparison between *ilk()* and *listP()*). Two forms of the *ilk()* command are used with lists:

ilk(variableName)
: This form of *ilk()* returns a *symbol* indicating the data type of the item, such as `#list`, `#propList`, `#rect`, or `#point` for the four list types. Note that linear lists return `#list`, not `#linearList`. There are dozens of possible return values when using *ilk()* with other data types, such as `#integer` and `#float`. See Table 5-4.

ilk(variableName, #dataType)
: This form of *ilk()* returns a *Boolean* indicating whether the item is of the specified data type. Note that `ilk(list, #list)` returns TRUE for all types of lists and is equivalent to `listP(list)`. Use `ilk(list, #linearList)` to check

only whether a list is a *linear* list. There are many symbols against which you can check an item's data type, as shown in Table 16-1.

With the exception of `#list` and `#linearList`, note that these two forms of *ilk()* are equivalent when used with lists:

```
if ilk(list) = #dataType then ...
```

or:

```
if ilk(list, #dataType) = TRUE then...
```

For example, to check if a list is a property list, use either:

```
if ilk(list) = #propList then ...
```

or:

```
if ilk(list, #propList) = TRUE then...
```

Table 6-6 shows the return values of the *ilk()* function. The second column shows the symbolic values returned by the *ilk(`list`)* form, and the remaining columns show the Boolean values returned by the *ilk(`list`, `#dataType`)* form.

Table 6-6: Ilk() Return Values

Data Type	ilk()	#list[1]	#linearList	#propList	#point	#rect
linear list	#list	TRUE	TRUE	FALSE	FALSE	FALSE
property list	#propList	TRUE	FALSE	TRUE	FALSE	FALSE
point	#point	TRUE	FALSE	FALSE	TRUE	FALSE
rect	#rect	TRUE	FALSE	FALSE	FALSE	TRUE
Nonlist	other	FALSE	FALSE	FALSE	FALSE	FALSE

[1] Same results a list().

Linear List Operations

Table 6-7 details operations for linear lists. You can also use *count()*, *min()*, *max()*, and *duplicate()* in the standard ways. Sorted linear lists are sorted by value.

Table 6-7: Linear List Operations

To Do This	Use This	Notes
Initialize a linear list	[ ], list(), duplicate(), value(string())	The list must be initialized before using any other list functions.
Determine if list is a linear list	if *ilk* (*list*, *#linearList*) or if *ilk*(*list*) = *#list*	Note different symbol names for the two forms of *ilk()*. *ilk(`list`)* never returns #linearList. See Table 6-6.

Table 6-7: Linear List Operations (continued)

To Do This	Use This	Notes
Add a value in sorted order	*add*(`list`, `val`)	Adds item to end of unsorted list or in order for sorted list.
Add a value at a specific position	*addAt*(`list`, `index`, `val`), or *add*(`list`, `index`, `val`)[1]	Pads missing elements with zeroes. Unsorts sorted lists.
Add a value to the end of the list	*append*(`list`, `val`), or *add*(`list`, `val`)	`Append` unsorts a previously sorted list. `Add` only appends to unsorted lists.
Replace an element by position	*setAt*(`list`, `index`, `val`), or *setaProp*(`list`, `index`, `val`)	Element is added if it doesn't already exist. Pads missing elements with zero.
Delete elements by position or value.	*deleteAll*(`list`), *deleteAt(`list`*, `index`), or *deleteOne*(`list`, `val`)[2]	Deleting elements changes indices of subsequent elements.
Get the value at a specific position or last position	*getAt*(`list`, `index`), or *getaProp*(`list`, `index`), or *getLast*(`list`)	*getAt()* and *getaProp()* are identical for linear lists.
Find the position of a specific value	*getOne*(`list`, `val`)[2] or *getPos*(`list`, `val`)[2]	Returns position of first match or zero if no match
Sort items by value	sort(`list`)	Commands that append data unsort the list.

[1] Undocumented variation of add() function.
[2] Case-sensitive

Property List Operations

Table 6-8 details operations for property lists. You can also use *count()*, *min()*, *max()*, and *duplicate()* in the standard ways. There is no *addPropAt()* command. Properties are added to the end of unsorted property lists or sorted by property name in sorted property lists.

Table 6-8: Property List Operations

To Do This	Use This	Notes
Initialize a property list	[:], duplicate(), value(string())	The list must be initialized before using any other list functions.
Determine if list is a property list	ilk (list, #propList) or ilk(list) = #propList	Correct symbol is *#propList*, not *#propertyList*. See Table 6-6.
Add a *property:value* pair	addProp(`list`, `prop`, `val`) or setaProp(`list`, `prop`, `val`)	*addProp()* adds item to end of unsorted list or in order for sorted list. *setaProp()* adds the property if it does not already exist.

Table 6-8: Property List Operations (continued)

To Do This	Use This	Notes
Replace a property's value by its property name	setaProp(*list*, *prop*, *val*) or setProp(*list*, *prop*, *val*)	*setaProp()* adds the property if it does not already exist. *setProp()* causes error if property does not exist.
Replace a property's value by position	setAt(*list*, *index*, *val*)	*setAt()* causes error if index is out of range. It does not add elements when used with property lists.
Delete elements by position or value	deleteAll(*list*), deleteAt(*list*, *index*), deleteOne(*list*, *val*)[1]	An element's *property:value* pair is always deleted as a unit.
Get the value of a specific property	getaProp(*list*, *prop*), or getProp(*list*, *prop*), or put the *prop* of *list*	If property is missing, *getaProp()* returns VOID, whereas *getProp()* and *put the prop of list* cause errors.
Get the name of the property at a specific position	getPropAt(*list*, *index*),	Index must be within range.
Get the value at a specific position or last position.	getAt(*list*, *index*) or getLast(*list*)	Index must be within range.
Get the first property name with a specific value	getOne(*list*, *val*)[1]	*getOne()* is case-sensitive! Returns property name or zero if not found.
Find the position of the nearest property name matching input	findPosNear(*list*, *prop*)[1]	Works best with sorted lists. See footnote 2 to Table 6-4 for return values.
Find the position of a specific property	findPos(*list*, *prop*)[1]	Returns index number or VOID.
Find the position of a specific value	getPos(*list*, *val*)[1]	Returns index number of first match or zero if not found.
Sort items by property name	sort(*list*)	Commands that append data, unsort the list.
Sort items by value	Can't be done automatically for property lists	Use a linear list, or sort it manually instead.

[1] Case-sensitive

Rect and Point Operations

Lingo's list commands interact with rects and points in bizarre ways. Some commands treat rects and points as linear lists, and others treat them as property lists.

A point can be thought of as a list of the form point(*x*, *y*) = [#locH:*x*, #locV:*y*].

A rect can be thought of as a list of the form rect(*l, t, r, b*) = [#left:*l*, #top:*t*, #right:*r*, #bottom:*b*]. A rect can also be specified as rect(point(*l, t*), point(*r, b*), but it is immediately converted to the rect(*l, t, r, b*) form.

The embedded properties of points and rects can be shown with this utility that reads the property names from any list.

Example 6-20: Extracting Unknown Properties from a List

```
on readProps listObj
  set objectType = ilk (listObj)
  put "This #" & objectType && "has" && [LC]
            count (listObj) && "properties"
  repeat with x = 1 to count (listObj)
    set thisProp = getPropAt (listObj, x)
    set thisValue = getaProp (listObj, thisProp)
    put "#" & thisProp & ":" && thisValue
  end repeat
end readProps

readProps(the clickLoc)
-- "This #point has 2 properties"
-- "#locH: 114"
-- "#locV: 83"

readProps(the rect of the stage)
-- "This #rect has 4 properties"
-- "#left: 0"
-- "#top: 0"
-- "#right: 160"
-- "#bottom: 120"
```

The *readProps()* utility in Example 6-20 can read properties from a *child object* or *Behavior instance*, too. See Chapter 12.

The names *#locH* and *#locV* (for points), and *#left*, *#top*, *#right*, *#bottom* (for rects) are generally the *only* valid property names for use with list commands requiring a property name. You can use them as:

```
set the locH of myPoint = 17
set the top of myRect = 52
```

Rects also support *reading* two additional properties; `height` and `width`

```
put the width of myRect
set someVar = the height of myRect
```

Table 6-9 details the list operations for rects and points. You can also use *count()*, *min()*, *max()*, and *duplicate()* in the standard ways. Note that rects and points are

generally not affected in a useful way by commands that add or delete elements, but they do work with commands that get and set the value of elements.

Table 6-9: Rect and Point Operations

To Do This	Use This	Notes
Define a rect	*rect*(`l, t, r, b`), or *rect*(*point*(`l, r`), *point*(`t, b`))	Rects always have four elements and cannot be defined with *list()*, [ ], or [:].
Define a point	*point*(`x, y`),	Points always have two elements and cannot be defined with *list()*, [ ], or [:].
Determine if list is a rect	*ilk*(`rect`) = #rect or *ilk* (`rect`, #rect) = TRUE	*ilk (rect,* `#list`*)* returns `TRUE`.
Determine if list is a point	*ilk*(`point`) = #point, or *ilk* (`point`, #point) = TRUE	*ilk (point,* `#list`*)* returns `TRUE`.
Add a value	Can't be done. Rects and points ignore the *add()*, *addAt(),* and *append()* commands and *addProp()*causes an error.	Points always have two elements, and rects always have four.
Replace a value by property name	*setaProp*(`rectOrPoint`, `prop`, `val`), or set the `prop` of `rectOrPoint`	Don't use *setProp()*. It causes errors in all cases, even when using "standard" rect and point properties.
Replace a value by position	*setAt*(`list`, `index`, `val`)	Use an index within range. (1-2 for points, 1-4 for rects).
Delete elements	Can't be done. *deleteAll()*, *deleteAt()*, and *deleteProp()* are ignored.	*deleteOne()* corrupts the point or rect structure.
Get the value of a specific property (#left, #top, #right, #bottom, #locH, or #locV)[1]	*getaProp*(`rectOrPoint`, `prop`) or put the *prop* of `rectOrPoint`[1]	Don't use *getProp()*. It causes errors in all cases, even when using "standard" rect and point properties.
Get the value at a specific position, or last position	*getAt*(`rectOrPoint`, `index`) or *getLast*(`rectOrPoint`)	Index must be within range.
Find the position of a specific property	The order of properties for rects and points is fixed.	Rects are always in the order (#left, #top, #right, #bottom). Points are always in the order (#locH, #locV).
Find the position of a specific value	*getPos*(`rectOrPoint`, `val`), or *getOne*(`rectOrPoint`, `val`)	Returns index number of first match or zero if no match.
Sort items by value or property name	Can't be done for points and rects.	Points and rects ignore the *sort()* command.

[1] You can also use *put the width of rect*, and *put the height of rect*, but you cannot set these properties.

Looping with Lists

There are two ways to extract each element in a list (see "*Repeat Loops*"in Chapter 1 for an overview of Lingo repeat loops). The following work for both linear lists and property lists.

You can loop for the number of elements returned by *count()*, using an integer index to extract data from the list. This gives you complete control, especially if the number of elements in the list is changing. The index variable (`i`) can also be used for other purposes, such as printing out an element's number.

Example 6-21: Printing Each Element's Position in a List

```
on showList someList
  set numElements = count (someList)
  repeat with i = 1 to numElements
    put "Element number" && i && "is" && getAt(someList, i)
  end repeat
end showList
```

Note that we counted the number of elements only *once*, rather than every time through the loop. If adding or deleting elements within the loop, you must instead recalculate the number of elements each time.

You can also use the *repeat with...in* syntax, which automatically extracts each value from the list.

Example 6-22: Printing Each Element's Value Only

```
on showList2 someList
  repeat with listValue in someList
    put "The value is" && listValue
  end repeat
end showList2
```

Note in Example 6-22 that `listValue` is the *value* of the current element, not an *index* (as was `i`). See "*Repeat Loops*" in Chapter 1 for important additional tips for working with the *repeat with...in* command.

List Utilities

Randomized Lists

The following handler creates a nonrepeating random list of `n` numbers (Refer to Chapter 8 for details on the *random()* function.) The trick is to place the numbers 1 through `n` into random locations throughout the list as it is built.

Example 6-23: Creating a List of Random Numbers

```
on randomNumberList n
  -- Initialize an empty list
  set myList = [ ]
  -- Create a list with the requested number of elements
```

Example 6-23: Creating a List of Random Numbers (continued)

```
  repeat with x = 1 to n
   -- Insert the next number in a random place in the list
    addAt (myList, random(x), x)
  end repeat
  return myList
end randomNumberList

put randomNumberList (10)
-- [8, 10, 7, 5, 1, 3, 9, 4, 2, 6]
```

The following *incorrect* routine (which you'll often see used) will likely result in some numbers being repeated in the list. It incorrectly adds random numbers that may already exist in the list because the *random()* function may generate the same number multiple times. See "*How Random Is Random?*" in Chapter 8 for details.

Example 6-24: Incorrect (Nonrandom) List Creation

```
on incorrectRandom n
  -- Initialize an empty list
  set myList = [ ]
  -- Create a list with the requested number of elements
  repeat with x = 1 to n
  -- Add a random number to the list
    add (myList, random(n))
  end repeat
  return myList
end incorrectRandom

put incorrectRandom  (10)
-- [9, 1, 6, 4, 4, 10, 2, 3, 9, 4]
```

This handler returns a randomized version of any linear list or property list, leaving the original list intact. Note that this randomizes an *existing* list, which differs from creating a random list, as shown above.

Example 6-25: Randomizing an Existing List

```
on randomizeList inList
  -- Exit if a valid list is not passed in
  if not listP(inList) then
    return void
  end if

  -- Initialize any empty list of the correct type
  if ilk (inList, #propList) then
    set outList = [:]
    set propList = TRUE
  else
    set outList = [ ]
    set propList = FALSE
  end if
```

Example 6-25: Randomizing an Existing List (continued)

```
  -- Work with a separate copy of the list
  set work = duplicate (inList)
  -- Loop while there are still elements in the list
  repeat while count(work)
    -- Select an element at random
    set next = random (count(work))
    if propList then
    -- Add the property element to the output list
      addProp (outList, getPropAt(work, next), ¬
                        getAt (work, next))
    else
      -- Add the linear element to the output list
      add (outList, getAt (work, next))
    end if
    -- Delete the element just selected
    -- so that it can't be used again
    deleteAt (work, next)
  end repeat

  return outList
end randomizeList
```

Nonrepeating Random Numbers in a Range

Suppose you want to simulate a game of Bingo. You'll need to generate a nonrepeating list of items that are chosen at random from the pool of remaining "game pieces."

One approach is to create a *randomized* list and then step through it *sequentially*. You can retraverse the list to repeat the "random" pattern.

An alternate technique is to create a *sequential* list, but *randomly* choose an item which you then *delete* from the list. (We'll use this technique in Chapter 8 to simulate a deck of cards.) The latter approach requires us to rebuild the list when it is emptied.

Suppose you want to create a randomized slide show. You may want to rerandomize the list of slides each time through *and* prevent the same slide from being shown twice in a row (if the last item from the previous round is the first item chosen in the next round). The following handler returns a *nonrepeating* random number from 1 to *n*. Every number from 1 to *n* is used once before any number is used twice, and the same number is never returned twice in a row. The key is to store the last number in a global variable and perform a lookup using *getPos()* to delete it from the newly regenerated list.

Example 6-26: A Nonrepeating Random Number Generator

```
on getUniqueRandomNumber n
  global gList, gLastNumber

  if voidP(gList) then set gList = randomNumberList(n)
```

Example 6-26: A Nonrepeating Random Number Generator (continued)

```
  if count (gList) = 0 then
    -- Recreate the list
    set gList =  randomNumberList(n)
    -- Locate and delete the last number from the new list
    set pickIt = getPos (gList, gLastNumber)
    deleteAt (gList, pickIt)
  end if

  -- Pick a random number and delete it from the list
  set pickIt = random (count(gList))
  set gLastNumber = getAt (gList, pickIt)
  deleteAt (gList, pickIt)
  return gLastNumber
end getUniqueRandomNumber
```

Converting List Types

Example 6-27 creates a linear list from a property list by stripping off the property names. It also works with rects and points.

Example 6-27: Converting a Property List to a Linear List

```
on convertToLinearList inList
  set outList = [ ]
  repeat with x in inList
    add outList, x
  end repeat
  return outList
end convertToLinearList

put convertToLinearList ([#fee: 3, #fie: 5, #fo: 6, #fum: 2])
-- [3, 5, 6, 2]
```

Example 6-28 builds a property list from two linear lists. The first list contains property names, and the second list contains values. The two lists must be of equal length.

Example 6-28: Creating a Property List from Two Linear Lists

```
on linearListToPropList symbolList, valueList
  set outList = [:]

  if not listP (symbolList) or not listP (valueList) then
    alert "This handler requires two lists"
    return VOID
  else
    set elements = count(valueList)
    set symbolCount = count (symbolList)
    if elements <> symbolCount then
      alert "Both lists must have the same length"
      return VOID
    end if
```

Example 6-28: Creating a Property List from Two Linear Lists (continued)

```
  end if

  repeat with n = 1 to elements
    addProp outList, getAt(symbolList, n), getAt(inList, n)
  end repeat
  return outList
end linearListToPropList

linearListToPropList ([#fee, #fie, #fo, #fum], [3,5,6,2])
put the result
-- [#fee: 3, #fie: 5, #fo: 6, #fum: 2]
```

This example could be modified to use the *createPropList()* handler from Example 6-1. That would allow people lacking square brackets on their keyboards to create property lists from two linear lists defined with the *list()* function.

Congratulations, you are now a *ListMeister* as well as a Lingo God!

Other Lingo Commands That Use Lists

Many Lingo commands and properties return or require linear lists, property lists, rects, and points. Refer to Chapter 5 in *Director in a Nutshell* for a detailed list of commands using rects and points.

Linear Lists

The following Lingo commands use true Lingo linear lists (not property lists). These can be manipulated like any other Lingo list, although many can be only read, not set.

the alertHook[1] *(can assign to a list of script instances)*
the actorList (list of objects)
the castMemberList of member[1] *(list of members for Custom Cursor Xtra)*
the cuePointNames of member[1] *(list of strings)*
the cuePointTimes of member[1] *(list of integers)*
the cursor of sprite (list of one or two 1-bit cast members)
cursor (list of one or two 1-bit cast members)
the deskTopRectList (list of rects)
the scoreSelection (list of lists that has changed format in D6)
the searchPath (list of strings)
the searchPaths (list of strings)
the selection of castLib (list of lists)
the windowList (list of windows)
the scriptInstanceList of sprite[1] *(list of script instances)*

1 New in D6

Property Lists

The following Lingo functions use true Lingo property lists (not linear lists). These lists can be manipulated like any other Lingo list.

These two handlers both return property lists that describe the parameters used in a Behavior. The author of a Behavior must define these lists in these handlers. Director asks for them when needed (see Chapter 12).

on getPropertyDescriptionList (returns list use to create dialog)
on runPropertyDialog (returns list of custom values for properties)

The UI Helper Xtra included with D6.5 provides a *getBehaviorInitializers()* function that returns the property list passed to *on runPropertyDialog*.

The MUI Dialog Xtra is a veritable orgy of property lists and nested lists. Refer to Chapter 15, *The MUI Dialog Xtra*, and to the downloadable Chapter 21, *Custom MUI Dialogs*, for details on these commands that use property lists:

Alert()
GetItemPropList()
GetWidgetList()
GetWindowPropList()
Initialize()
ItemUpdate()

Pseudo-Lists

There are several Lingo commands that don't use Lingo lists but use *pseudo-lists* that you might confuse with, or wish to convert to, true Lingo lists.

Child Objects Properties

Although child objects are not identical to property lists, they can also contain properties that can be manipulated like a property list using the *count()*, *getAt()*, *getPropAt()*, and *setaProp()* commands. Refer to Chapter 12 for details. The properties of a child object can be extracted using the *readProps()* handler in Example 6-20.

Text Lists and Pick Lists

These commands don't use true Lingo lists but rather use text strings that are list-like:

item of, word of, line of, char of chunk expressions (see Chapter 7)
the labelList (see Example 6-29)
mMessageList (see Chapter 13)
xFactoryList (see Chapter 13)

For example, *the labelList* returns a text string with each marker label name on a separate line. Such strings can not be used with the list commands but can be parsed as described under "*Text Parsing*" in Chapter 7. It is, however, fairly trivial

to convert a text string delimited by carriage returns (as is *the labelList*) to a true Lingo list.

Example 6-29: Converting a Text String to a Lingo List

```
on convertLinesToList textList
  set realList = []
  -- Convert a list from a text string
  -- to a true Lingo list
  repeat with x = 1 to the number of lines in textList
    add (realList, line x of textList)
  end repeat
  return realList
end convertLinesToList
```

The output below will depend on your *labelList*:

```
put convertLinesToList (the labelList)
-- ["menu1", "menu2", "New Marker"]
```

The generalized handler above can extract lines of text from *any* string:

```
put convertLinesToList ("Oh" & RETURN & "Atlanta")
-- ["Oh", "Atlanta"]
```

Refer to the example under "*Text Parsing*" in Chapter 7 that parses all types of strings, not just those with multiple lines delimited by carriage returns.

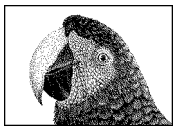
Don't confuse Lingo lists with text lists from which the user can choose an item. There is no way for the user to "see" a Lingo list unless its contents are copied to a field cast member. Use field cast members themselves for user interaction.

Refer to Chapter 12, *Text and Fields*, in *Director in a Nutshell*.

Variable-Length Parameter Lists

Lingo handlers can accept any number of parameters. The Lingo commands *param()* and *the paramCount* are list-like functions that decipher the parameters passed into a handler. *The paramCount* represents the total number of parameters, and *param(n)* returns the n^{th} parameter, analogous to the *count()* and *getAt()* list functions. Refer to "*Parameters and Arguments*" in Chapter 1 for a detailed discussion of using *the paramCount* and *param()* to decipher variable-length parameter lists.

Likewise, the *externalParamCount()*, *externalParamName()*, and *externalParamValue()* functions are used to access a "list" of the parameters specified in the HTML tag used to embed a Shockwave movie in an HTML page. Refer to Chapter 11, *Shockwave and the Internet*, in *Director in a Nutshell* for details.



CHAPTER 9

Mouse Events

Mouse Events

Let's explore Director's sometimes counter-intuitive handling of mouse-related events. (See Chapter 2, *Events, Messages, and Scripts,* for details on Director's overall event handling and script creation.) This chapter is about understanding mouse event processing, not about creating buttons per se. Refer to Chapter 14, *Graphical User Interface Components,* in *Director in a Nutshell* for details on custom cursors and on creating well-behaved buttons and using the new Custom Button Editor Xtra. See also Example 4-6. "Manipulating Sprite Properties to Add Interactivity to a Button" in Chapter 4, *CastLibs, Cast Members, and Sprites*, in *Director in a Nutshell.*

Whenever the mouse button is pressed or released, Director generates *mouseUp*, *mouseDown*, *rightMouseUp*, or *rightMouseDown* events. Attach scripts with matching *event handlers* to turn the sprite into a clickable button.

Note that for simple linear presentations, the Tempo channel can be used to wait in a frame for a mouse click (or key press) before advancing the playback head. In prior versions of Director, the Tempo channel would ignore mouse events while waiting for time to elapse or for a sound or digital video to play. This limitation has been removed in Director 6 (although the Tempo channel doesn't work in Shockwave).

The new *mouseEnter*, *mouseLeave*, and *mouseWithin* events are sent when the cursor enters or leaves a sprite with a corresponding handler attached (no mouse click is required). Leave *the idleHandlerPeriod* at its default setting (0) when using these events.

Suppose you have two overlapping sprites in channels 1 and 2, as shown in Figure 9-1. The *top-most* sprite in the higher channel (channel 2) with be in the foreground, and the sprite in the lower channel (channel 1) will be *obscured* (covered partially by the sprite in channel 2). (If I say that sprite 1 is *below* sprite 2, I am referring to their appearance on the Stage, not their relative positions in the Score window sprite channels.)

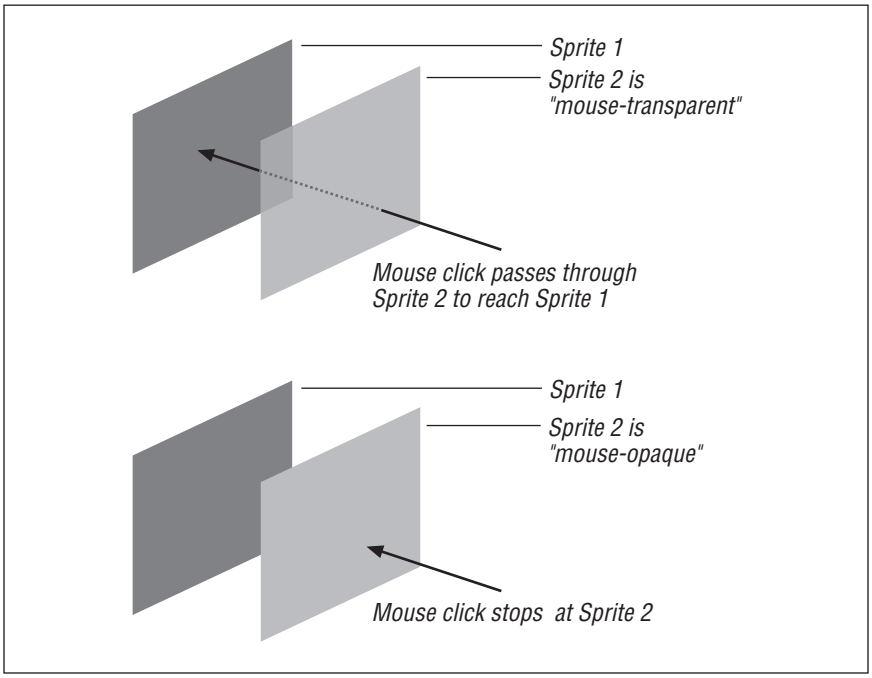


Figure 9-1: Two overlapping sprites

Create two rectangular bitmap sprites in the Paint window and overlap them partially on Stage, as shown in Figure 9-1. Attach the following sprite script to the sprite in channel 1 (not channel 2).

Example 9-1: Basic Mouse Events

```
on mouseUp
  put "MouseUp detected by sprite" && the clickOn
end
```

Set the movie to loop using the Control Panel, then rewind and play the movie. If you click on the top-most sprite in channel 2 (which has no script) in the region where it overlaps sprite 1, you should see the following in the Message window:

```
-- "MouseUp detected by sprite 1"
```

Note that the event registers for sprite 1, not sprite 2, because Director generally ignores sprites that don't have scripts attached. As far as Director is concerned, the top-most sprite is *mouse-transparent* (mouse clicks pass right through it).

Now attach the following *mouseDown* script to the sprite in channel 2.

```
on mouseDown
  put "MouseDown detected by sprite" && the clickOn
end
```

Rewind and play the movie and click on the top-most sprite again. Notice that the *mouseDown* handler in sprite 2 not only intercepted the *mouseDown* event, it *also* prevented the *mouseUp* event from reaching the obscured sprite behind it in channel 1.

Sprites intercept events that occur within their boundary. Use the *matte* ink to respond only to events within the irregular outline of the sprite. When other inks are used, the sprite responds to mouse events within its entire bounding rectangle.

Mouse-Opaque Sprites

It is important to understand mouse event handling clearly because it is at the heart of every Director project—and many programmer bugs. Macromedia uses the term *mouse-opaque* in the D6.0 *ReadMe* file to describe sprites that intercept mouse click events as described previously.

A sprite that has an *on mouseUp*, *on mouseDown*, *on rightMouseUp*, or *on rightMouseDown* handler attached becomes *mouse-opaque* and prevents *all four* of those mouse events from passing through to any sprites beneath it.

You can think of a mouse-opaque sprite as a sponge that absorbs mouse clicks or as an umbrella that prevents mouse events from landing on any sprites below. A *mouse-transparent* sprite can be thought of as a piece of clear glass that casts no shadow. It neither detects the events nor prevents them from reaching other sprites. In Director 5, *any* script attached to a sprite turned the sprite mouse-opaque (that is, would prevent underlying sprites from receiving mouse events). In Director 6, this was changed so that only those four handlers named previously cause a sprite to become mouse-opaque. This allows sprites to react to new messages, such as *beginSprite*, without necessarily being mouse-opaque. To force a sprite to be mouse-opaque in Director 6, you can attach a dummy script that has *only* a comment in it (and no handlers declared), such as:

```
-- This dummy script makes the sprite mouse-opaque
```

This makes the sprite mouse-opaque without actually trapping one of the four special mouse click events. See text that follows for details on *the clickOn*, which works only for sprites that have a script attached.

Mouse-opaque sprites prevent the four mouse click messages only from being sent to *other sprites*. They don't prevent untrapped mouse events from passing to frame scripts or movie scripts, nor do they prevent other scripts attached to the *same* sprite from receiving events.

For example, even if a sprite has a *mouseDown* handler attached, it prevents only *mouseDown* events from, say, reaching the frame script. Any *mouseUp* events would still reach the frame script (if the latter had a *mouseUp* handler). See Chapter 2 for details on how events are usually passed through the script hierarchy.

If a *mouseUp*, *mouseDown*, *rightMouseUp,* or *rightMouseDown* handler passes an event with the *pass* command, the event is not forwarded to other sprites. It passes first to the sprite's cast script (if any) and then on to the frame script and possibly the movie script. You must manually broadcast the mouse click message using *sendSprite* or *sendAllSprites* for it to reach multiple sprites.

This discussion of mouse-opaque sprites pertains only to mouse *click* events. Other types of mouse cursor events are discussed next.

Cursor-Opaque Sprites

The mouse-opacity of a sprite has no effect on how the new *mouseEnter*, *mouseLeave,* and *mouseWithin* events are handled. I've coined the name *cursor-opaque sprites* to describe Director's special handling of these new events based on the cursor position (not mouse clicks). Again, suppose you have two overlapping sprites (see Figure 2-1), and the bottom (partially obscured) sprite has *mouseEnter*, *mouseLeave,* and *mouseWithin* handlers attached. If the top-most sprite does not have one of these three handlers attached, too, it is *cursor-transparent* and is completely ignored by Director in regard to cursor-related events. Thus, if the cursor enters the top-most sprite but is still within the *obscured* sprite's boundaries, the obscured sprite continues to receive *mouseWithin* messages. *MouseEnter* or *mouseLeave* messages are *not* generated as the cursor enters or leaves the overlapping area of the two sprites. It is as if the top-most sprite didn't even exist.

Now if you add a *mouseEnter*, *mouseLeave, or mouseWithin* handler to the top-most sprite, it will become *cursor-opaque* and be "seen" by Director cursor events. Rolling into the overlap area will now send *mouseWithin* events to the top-most sprite, not the bottom sprite. Paradoxically, even though the top-most sprite now prevents *mouseWithin* events from reaching the obscured sprite, rolling between the two overlapped sprites now generates *mouseEnter* and *mouseLeave* events for *both* sprites. The second cursor-opaque sprite acts as a "foil" (in the dramatic sense) for the first sprite.

A cursor-opaque sprite receives a *mouseLeave* event when it becomes invisible or is moved off-stage.

Custom Buttons created with the Button Editor Xtra ignored *mouseEnter*, *mouseWithin*, and *mouseLeave* events in D6.0, but this problem was fixed in D6.0.1.

Checking for Mouse Events

The sections below explore the question, "Who am I?". Use them to write generalized Behaviors that work regardless of the sprite to which they are attached.

Director automatically dispatches events to the correct sprite. You need not attempt to determine which sprites were clicked from a frame script. Instead, attach appropriate mouse event handlers directly to the sprites of interest.

Frame scripts that manually check which sprite was clicked are considered extremely poor style in Director. The following approach should be avoided:

```
on exitFrame
  if the clickOn = 1 then
    -- Do action for sprite 1
  else if the clickOn = 2 then
    -- Do action for sprite 2
  end if
end
```

Instead, attach, say, *on mouseUp* handlers to each sprite of interest, as in:

```
on mouseUp me
  -- Perform some action for this sprite
end
```

See the tip under "*Determining Which Sprite Was Rolled Over*" regarding the analogous issue with rollover-related events.

Determining the Current Sprite's Number

In Chapter 1, *How Lingo Thinks,* we discussed generalizing a handler to make it more flexible. To reiterate, you should avoid "hardcoding" sprite channel numbers, such as:

```
on mouseUp
  set the foreColor of sprite 2 = random (255)
end
```

If possible, you should instead use a system property to determine the sprite's channel number automatically, so that the script will work for any sprite to which it is attached, even if you move the sprite to another channel.

In the "*To Me or Not To Me"* section of Chapter 2, we saw how *the currentSpriteNum* and *the spriteNum of me* always indicate the sprite channel number to which the script is currently attached.

Example 9-2: What's My Sprite Number?

```
on mouseUp
  -- Set the current sprite's foreColor to a random color
  set the foreColor of sprite (the currentSpriteNum) = ¬
    random (255)
end
```

To use *the spriteNum of me* property, you must declare *me* as a parameter following the handler name as shown:

```
on mouseUp me
  -- Declare "me" (see above) when using "the spriteNum of me"
  set the foreColor of sprite (the spriteNum of me) = ¬
    random (255)
end
```

Determining Which Sprite Was Clicked

Use *the clickOn* property or *clickOn()* function to determine the last sprite that the user clicked.

The clickOn. property and *clickOn()* function ignore sprites without scripts.

If you click a scriptless sprite, *the clickOn* will register for the first sprite beneath it that has a script attached. If no sprites under the mouse click have scripts, *the clickOn* returns 0, as it does if you click on the Stage.

From within the *mouseUp* or *mouseDown* handler of the sprite that was clicked, *the clickOn* would be the same as *the currentSpriteNum.* In prior versions of Director you will often see Lingo code such as:

```
on mouseUp
  set the foreColor of sprite (the clickOn) = random (255)
end
```

The spriteNum of me and *the currentSpriteNum* were added in Director 6 because sprites receive many new events that do *not* require mouse clicks. They are useful from within *beginSprite*, *endSprite*, *mouseEnter*, *mouseLeave,* and *mouseWithin* handlers, as well as *mouseUp* and *mouseDown* handlers. The *SpriteNum* of me is useful only from within script scripts (where it returns the current sprite's number); from within cast scripts (which don't receive the *me* instance) use the *CurrentSpriteNum* instead. From within frame scripts and movie scripts *the current SpriteNum* and *the SpriteNum of me* are not meaningful.

However, *the clickOn* returns the last sprite clicked even when checked from a frame script, movie script, or different sprite script. (*The clickOn* and *clickOn()* do not update within a handler. The value returned is that obtained from before the handler was called.)

The clickLoc property (or *clickLoc()* function) returns the point where the mouse was last clicked, relative to the upper left corner of the Stage. The mouse may have moved since it was clicked, so check *the clickLoc* rather than *the mouseH* or *the mouseV*. (*The clickLoc* does not update within a handler. The value returned is that obtained from before the handler was called.)

Determining Which Sprite Was Rolled Over

Director 6 vastly simplifies rollover detection. Simply attach *mouseEnter* and *mouseLeave* handlers that highlight and unhighlight the button of interest (say, by swapping and restoring its *member of sprite* property). In prior versions of Director, you must use the *rollover()* function from within a frame script or movie script.

The rollover property and *rollover()* function (without any argument) return the number of the sprite being rolled over (or 0 when rolling over the Stage). The *rollover(n)* function (where `n` is a sprite number) returns a *Boolean* value indicating whether the cursor is over the specified sprite. Theoretically, if *the rollover* equals `n`, then *rollover(n)* should return `TRUE`, but the different forms of the *rollover* command disagree when the cursor is over the Stage. So *rollover(0)* is always `FALSE`, even though it should theoretically return `TRUE` when *the rollover* or *rollover()* returns 0.

Furthermore, the different forms of the *rollover* command may disagree when testing the rollover for overlapping or invisible sprites. *The rollover* returns the number of the top-most *visible* sprite. *Rollover(n)* will return `TRUE` even if sprite `n` is invisible or not the top-most sprite. But *the rollover* and *rollover()* forms of the command ignore invisible sprites. Either refrain from testing *rollover(n)* on invisible sprites, or move the sprite off-stage instead.

The mouseMember property returns *the member of sprite* property of the top-most visible sprite over which the mouse resides. You can check it against the *rollover(n)* value, such as:

```
on exitFrame
  if rollover(5) and [LC]
     (the mouseMember = the member of sprite 5) then
    put "We are apparently rolling over sprite 5"
  end if
end
```

This code is reliable only if the same cast member is not used for multiple overlapping sprites.

In Director 6, you rarely need to use *the rollover* property or *rollover()* function. Instead use *mouseEnter*, *mouseWithin,* and *mouseLeave* handlers attached to the sprites of interest to handle rollover conditions, or the Custom Button Xtra to handle button rollover states automatically.

Invisible Sprites

In Director 5, only mouse-related events were sent to sprites. In Director 6, sprites receive other new events unrelated to the mouse. In Director 5, setting *the visible of sprite* property to FALSE hid the sprite and disabled all of its handlers (which were all mouse-related). This was equivalent to muting the sprite's channel in the Score window. In Director 6, *the visible of sprite* property affects only mouse-related events for that sprite. The *beginSprite*, *endSprite*, *prepareFrame*, *enterFrame*, and *exitFrame* handlers can be disabled only by muting the sprite's channel in the Score window. (Muting disables all events for that sprite channel, but setting *the visible of sprite* affects only its visibility and mouse-related events).

Standard Mouse Events

In the simplest case, Director generates a *mouseDown* event when the user *depresses* the mouse button and a *mouseUp* event when the user *releases* the mouse button. A given script may contain handlers that trap all, some, or no mouse events.

You should generally trap either *mouseUp or mouseDown* events (see text that follows for caveats when trapping both events).

The *mouseDown* and *mouseUp* messages may be sent to *different* sprites possibly in different frames if the *mouseDown* handler moves the playback head. Trapping both messages can lead to two different sprites unintentionally responding to a single mouse click sequence. During the time between the two events, other messages, such as *mouseEnter*, may be generated. Table 9-1 lists mouse events in the order in which they are generated, *assuming the simplest case in which a user clicks a sprite*. In reality, multiple *mouseEnter*, *mouseWithin,* and *mouseLeave* events can be generated after the *mouseDown* event and before the *mouseUp* event. As the user moves the cursor among various sprites, many mouse rollover events may be generated without any mouse click events being sent.

The frequency of *mouseEnter*, *mouseLeave,* and *mouseWithin* events depends on *the idleHandlerPeriod*. Rollover events are unusably sluggish if *the idleHandlerPeriod*. is not set to 0.

The events shown in column 1 of Table 9-1 should not be confused with the properties in Table 9-5.

Table 9-1: Mouse Click and Roll Events

Event	Generated When or If:	Left	Right
mouseEnter[1]	Cursor enters a sprite's outline[2]	•	•
mouseWithin[1]	Cursor is within a sprite's outline[2] (sent *repeatedly*)	•	•
mouseDown	Left mouse button depressed	•	
rightMouseDown	Right mouse button depressed		•
mouseLeave[1]	Cursor leaves a sprite's outline[2]	•	•
mouseUpOutside[1,3]	Either mouse button released outside a sprite[2] after being depressed inside a sprite	•	•
mouseUp	Left mouse button released	•	
rightMouseUp[3]	Right mouse button released		•
buttonClicked[1,4]	Mouse is released (not pressed) over a Custom Button sprite	•	

[1] New in Director 6.0.
[2] See the "*Cursor-Opaque Sprites*" section earlier in this chapter. The *mouseEnter*, *mouseWithin,* and *mouseLeave* events obey the matte outline of bitmap sprites using the matte ink, but not for QuickDraw shape sprites. For other inks, the outline is the sprite's bounding box.
[3] There is no on *rightMouseUpOutside* event.
[4] The *buttonClicked* message is sent only if the *enabled of member* or *enabled of sprite* property of the custom button is TRUE. See Chapter 14 in *Director in a Nutshell*.

Table 9-2 shows how each mouse event is passed through the Lingo messaging hierarchy. A given event may be handled by a primary event handler, sprite script(s), cast script, frame script, or movie script. Refer also to the Table 2-7 in Chapter 2.

See the discussion of mouse-opaque and cursor-opaque events at the beginning of this chapter. Untrapped mouse click events are passed onto frame or movie scripts.

Use *rollover()*, which is a *function*, not an *event*, to check for rollovers from within frame and movie scripts

Table 9-2: Mouse Event Passing

Event/Message Type	Message Passing
mouseDown	mouseDownScript →Sprite scripts and/or cast scripts of mouse-opaque sprites →Frame script →Movie scripts.
mouseUp[1]	mouseUpScript →Sprite scripts and/or cast scripts of mouse-opaque sprites →Frame script →Movie scripts.
mouseEnter mouseLeave mouseWithin	Sprite scripts and/or cast scripts of cursor-opaque sprites. (See "*Cursor-Opaque Sprites*" earlier in this chapter.) These events are never passed to frame or movie scripts.
mouseUpOutside	Sprite scripts and/or cast scripts of sprite that received original *mouseDown* or *rightMouseDown* event. See "*MouseUpOutside Events*" later in this chapter.
rightMouseDown[2] *rightMouseUp*[1,2]	Sprite scripts and/or cast scripts of mouse-opaque sprites →Frame script →Movie scripts.
buttonClicked	Sent by Custom Buttons to sprite scripts and/or cast script of clicked sprite only. Not sent to obscured sprites or passed to Frame script or Movie script.

[1] The *mouseUp* and *rightMouseUp* outside events are sent directly to the frame script, bypassing the sprite and cast scripts, if they follow a *mouseUpOutside* event and *the buttonStyle* property is `TRUE`.
[2] There are no primary event handlers for the *rightMouseUp* and *rightMouseDown* events.

MouseUpOutside **Events and** *the ButtonStyle*

Director 6 generates a new event, *mouseUpOutside*, if the user depresses the mouse button over a mouse-opaque sprite and then subsequently releases the mouse button over a different sprite or no sprite. It is not generated if the user clicks first on the Stage or a non-clickable sprite.

There is no *rightMouseUpOutside* event. *MouseUpOutside* is generated whether the original mouse click was a *mouseDown* or *rightMouseDown* event. *MouseUpOutside* events are sent, if appropriate, before and in addition to (not instead of) *mouseUp* or *rightMouseUp* events. The system property *the buttonStyle* determines whether Director sends a *mouseUp* (or *rightMouseUp*) event following a *mouseUpOutside* event.

The *mouseUpOutside* event is sent only to the original sprite over which the *mouseDown* or *rightMouseDown* occurred. The second sprite, over which the cursor is released, receives the *mouseUp* or *rightMouseUp* event instead (only if *the buttonStyle* is `FALSE`, the default).

If *the buttonStyle* is `FALSE` (the default) a button will highlight as the mouse rolls over it, even if the mouse button was pressed over another sprite initially. (A bitmap sprite will only highlight if its *Hilight When Clicked* option is set in its Cast Member Properties dialog box. This option is unrelated to *the hilite of member* property. Mouse-opaque shape sprites always highlight when clicked.) When the mouse button is released, the sprite under the mouse will receive a *mouseUp* (or *rightMouseUp*) event. This is called *list style* in the *Lingo Dictionary* and is appropriate for buttons that make up a list of choices.

If *the buttonStyle* is `TRUE` (so called *dialog style*) only the initial button receiving the *mouseDown* will highlight. When the mouse button is released, the sprite under the mouse will *not* receive a *mouseUp* (or *rightMouseUp*) event, unless it was the original sprite. In this case, the *mouseUp* (or *rightMouseUp*) event is sent directly to the frame script (and possibly the movie script), bypassing any sprite scripts or cast scripts attached to the sprite over which the mouse was released.

Double-Clicks

The Boolean system property *the doubleClick* indicates whether the two last mouse clicks occurred within the time interval set in the Macintosh or Windows Mouse Control Panel. *The doubleClick* depends on the interval between the two *mouseDown* or *rightMouseDown* events, not the two *mouseUp* or *rightMouseUp* events. The Mac Finder is slightly different in that it requires that both the two *mouseDown* and two *mouseUp* events all occur in the prescribed interval. Under Windows, even if the two clicks involve the two different mouse buttons (right and left), they would still constitute a double-click. *The doubleClick* is automatically reset to `FALSE` when the elapsed interval between clicks exceeds the operating system's double-click setting.

There is no way to read or change the user-defined double-click interval from Lingo, but you can create your own double-click handler to simulate this. Likewise, you must manually detect triple-clicks, if so desired.

Consumer titles should avoid requiring the user to double-click the mouse. Especially in children's games, you should ignore rapid-fire events.

For example, if a single button is used to start and stop a song in a children's game, many children will click the button rapidly twice. This would start and then immediately stop the song. If the double-click interval is very short, checking *the doubleClick* property will not prevent spurious mouse clicks.

The doubleClick is a system property that can be checked, not an event that can be trapped. The first mouse event is always sent separately from the second mouse event that sets *the doubleClick*.

In Example 9-3, the *mouseUp* handler will be called *twice* if the user double-clicks—once with *the doubleClick* equal to `FALSE` and again with it equal to `TRUE`. Thus, the *singleClickAction* will be called first, before the user has double-clicked.

Example 9-3: Responding to Both Single- and Double-Clicks

```
on mouseUp
  if the doubleClick then
    doubleClickAction
  else
    singleClickAction
  end if
end
```

Solve the problem using the following handler described in detail in Macromedia's *Lingo Dictionary* or online *Help* under the *doubleClick* keyword entry. (Note that this example tramples any existing timers that use *the timer* property. See Chapter 11, *Timers and Dates*, for ways to avoid this.)

Example 9-4: Preventing a Double-Click from Executing the Single-Click Action

```
on mouseUp
  if the doubleClick then
    exit
  else
    startTimer
    repeat while the timer < 20
      -- Simulate check for double-click before
      -- deciding whether to perform singleClickAction
      if the mouseDown then
        doubleClickAction
        exit
      end if
    end repeat
    singleClickAction
  end if
end
```

Eating Rapid-Fire Clicks

The system property *the lastClick* is reset to 0 at the time of each click. Therefore, it reflects the time since the *current* click. It is not useful for preventing double-clicks unless you store the previous value in a variable and do some extra math, as follows. Example 9-5 prevents clicks closer than 60 ticks (1 second) apart from being allowed.

Example 9-5: Preventing Rapid-Fire Clicks

```
property pLastMouseClick

on mouseDown me
  if voidP(pLastMouseClick) then
    set pLastMouseClick = the ticks - 61
  end if
```

Example 9-5: Preventing Rapid-Fire Clicks (continued)

```
  if (the ticks - pLastMouseClick) < 60 then
    exit
  else
    set pLastMouseClick = the ticks
  end if
  -- Perform work here.
end
```

Use this to disable double-clicks entirely:

```
set the mouseUpScript = "if the doubleClick then stopEvent"
```

Multi-button Mice

Most projects use only the left mouse button under Windows and the single Macintosh mouse button, but Director supports two of the three Windows mouse buttons and can emulate a two-button mouse on the Macintosh.

In the authoring mode, Director uses the right mouse button under Windows and `Control-click` on the Macintosh for context-sensitive menus. Refer to Chapter 2, *Being More Productive,* in *Director in a Nutshell* for details on these shortcuts.

Note that the following commands ignore the right mouse button:

on mouseUp
on mouseDown
the mouseUpScript
the mouseDownScript

Conversely, the following *do* respond to, affect, or reflect the state of the right mouse button:

on mouseUpOutside
on rightMouseUp
on rightMouseDown
the clickOn
the doubleClick
the emulateMultiButtonMouse
the lastClick
the lastEvent
the mouseDown
the mouseUp
the rightMouseDown
the rightMouseUp
the stillDown

The following are not affected by the left mouse button:

on rightMouseUp
on rightMouseDown

the rightMouseDown
the rightMouseUp

Windows Right Mouse Button

Director for Windows supports the left and right mouse buttons. The middle button, if any, is always ignored. The Windows 95 Mouse Control Panel allows the user to configure the mouse for a right-handed user (*left* mouse button is the primary button) or a left-handed user (*right* mouse button is the primary button). For simplicity, Director uses the primary mouse button in the default case and refers to the secondary button as the "right" button. For a left-handed mouse setup, Director treats the physically rightmost button as the *primary* button, and events such as *rightMouseUp* and *rightMouseDown* would correspond to the user's physical left mouse button. In short, Director for Windows remaps the buttons properly, so you don't have to worry about the user configuration. For simplicity, I refer to the secondary mouse button as the "right" mouse button. Penworks (*http://www.penworks.com*) publishes a free utility, *SwapEm*, which dynamically reverses the right and left mouse buttons via Lingo.

Table 9-3 shows the events generated by each of the Windows mouse buttons and the simulated Macintosh mouse buttons. It also shows the properties that each action sets. Note that *mouseUpOutside* events may be generated for either mouse button. See "*Mouse Event Idiosynchrasies*" for possible errors when using multi-button mice.

Table 9-3: Mouse Button Events

User Action	**Message Sent**	**Also Sets Properties**
left mouse button click	mouseDown	the mouseDown the stillDown the clickOn the clickLoc the lastEvent the lastClick the doubleClick (if applicable)
left mouse button release	mouseUp	the mouseUp
left mouse released outside	mouseUpOutside	the mouseUp
right mouse button click	rightMouseDown	the rightMouseDown the mouseDown the stillDown the clickOn the clickLocthe lastEvent the lastClick the doubleClick (if applicable)

Table 9-3: Mouse Button Events (continued)

User Action	Message Sent	Also Sets Properties
right mouse button release	rightMouseUp	the rightMouseUp the mouseUp
right mouse released outside	mouseUpOutside	the mouseUp
middle mouse button click	<*none*>[1]	N/A
middle mouse button release	<*none*>[1]	N/A

[1] Some of the third-party Xtras cited at the end of this chapter can read the middle mouse button under Windows.

Note that your standard *mouseUp* and *mouseDown* handlers will never be triggered if the user clicks the right mouse button. You can remind him to click the left mouse button by placing a handler such as this in a movie script:

```
on rightMouseDown
  alert "Please use the left mouse button instead."
end
```

In practice, using the *alert* command can interfere with custom palettes. Refer to Chapter 13, *Graphics, Color, and Palettes,* and to Chapter 14 in *Director in a Nutshell.*

The system property *emulateMultiButtonMouse* and the state of the `Control` key are always ignored in regard to mouse events under Windows.

Macintosh Right Mouse Button

Macintosh mice typically have only one button. If the system property *emulateMultiButtonMouse* is `TRUE` (the default is `FALSE`), Director generates *rightMouseUp* and *rightMouseDown* events when you `Control`-click, as shown in Table 9-4. Note that the event type (right versus left) is determined when the mouse button is first pressed, regardless of whether the state of the `Control` key changes before the mouse button is released. That is, there is never a *rightMouseUp* event without a preceding *rightMouseDown* event, nor a *mouseUp* event without a preceding *mouseDown* event. Note that *mouseUpOutside* events may be generated in either case.

Table 9-4: Macintosh Mouse Button Mapping

User Action	emulateMulti-ButtonMouse	Control Key	Message Sent
mouse click	FALSE	N/A	mouseDown
mouse release	FALSE	N/A	mouseUp
mouse click	TRUE	Up	mouseDown
mouse release	TRUE	Up	mouseUp

Table 9-4: Macintosh Mouse Button Mapping (continued)

User Action	emulateMulti-ButtonMouse	Control Key	Message Sent
mouse click	TRUE	Down	rightMouseDown
mouse release	TRUE	Down	rightMouseUp
mouse release outside original sprite	N/A	N/A	mouseUpOutside

Mouse Properties

Numerous system properties provide information about the current mouse button state, the last mouse click event, and the current mouse position. Most of these events can be tested but not set. Table 9-5 summarizes all the system properties related to the mouse button state, and Table 9-6 summarizes the system properties related to the position of the mouse cursor.

Table 9-5: Mouse Button State System Properties

Property Name	Usage	Left Button	Right Button
the buttonStyle	Determines whether *mouseUp* and *rightMouseUp* event are sent to other sprites or only to the frame following a *mouseUpOutside* event.	•	•
the clickLoc	Point on stage where mouse was last clicked. See Chapter 5, *Coordinates, Alignment, and Registration Points,* in *Director in a Nutshell*.	•	•
the clickOn	Sprite number of last mouse-opaque sprite clicked, or 0 (zero) to indicate the Stage.[1]	•	•
the currentSpriteNum	Sprite number of the sprite to which the current Lingo handler is attached.[1]	•	•
the doubleClick	Boolean indicating whether last mouse event constituted a double-click.	•	•
the emulateMultiButtonMouse	Boolean indicating whether to treat `Control`-clicks as right mouse clicks on Macintosh.		•
the lastClick	Time (in ticks) since last mouse click. See Chapter 11.	•	•
the lastEvent	Time (in ticks) since last mouse click, mouse roll, or key press. See Chapter 11.	•	•
the mouseDown	Boolean indicating whether *either* mouse button (left or right) is depressed.	•	•
the mouseDownScript	Primary event handler for *mouseDown* events. Not called for *rightMouseDown* events.[2]	•	

Table 9-5: Mouse Button State System Properties (continued)

Property Name	Usage	Left Button	Right Button
the mouseUp	Boolean indicating whether *both* mouse buttons (left or right) are up.	•	•
the mouseUpScript	Primary event handler for mouseUp events. Not called for *rightMouseUp* events.[2]	•	
the rightMouseDown	Boolean indicating whether the right mouse button is depressed.		•
the rightMouseUp	Boolean indicating whether the right mouse button is up.		•
the spriteNum of me	Sprite number of the sprite receiving the message.[1]		•
the stillDown	Boolean indicating whether the mouse button is still depressed following a *mouseDown* or *rightMouseDown* event. Returns FALSE if mouse has been released in the interim.	•	•

[1] See "*Which Sprite Was Clicked On or Rolled Over?*" earlier in the chapter.
[2] Note there are no *rightMouseUpScript* or *rightMouseDownScript* properties.

The properties in Table 9-6 change when the cursor moves, regardless of the state of the mouse button(s).

Table 9-6: Mouse Location/Movement System Properties

Property Name	Usage
cursor `whichCursor`	Changes cursor when cursor is over the Stage. See Chapter 14 in *Director in a Nutshell*.
the cursor of sprite `whichSprite`	Changes cursor when cursor is over a particular sprite. See Chapter 14 in *Director in a Nutshell*.
the lastEvent	Time (in ticks) since last mouse click, mouse roll, or key press.
the lastRoll	Time (in ticks) since last mouse movement.
the immediate of sprite `whichSprite`	Obsolete; required in Director 3.1.3 to detect mouseDown events.
the mouseCast[1]	Obsolete; use *the mouseMember* instead.
the mouseMember[1]	Cast member over which cursor is located (even sprite's without scripts).
the mouseChar[2]	Character position under cursor in field sprite.
the mouseH	Cursor's horizontal position relative to left edge of the Stage.

Table 9-6: Mouse Location/Movement System Properties (continued)

Property Name	Usage
the mouseItem[2]	Item number under cursor in field sprite. See *the itemDelimiter*.
the mouseLine[2]	Line number under cursor in field sprite.
the mouseV	Cursor's vertical position relative to top of Stage.
the mouseWord[2]	Word number under cursor in field sprite.
the preLoadEventAbort	Determines whether a mouse event aborts preloading.
rollover (*n*)	Boolean function, returning TRUE if cursor is over sprite number *n*., even if it is invisible, has no scripts, or is obscured by other sprites.[1]
the rollover rollover()	Indicates the top-most mouse-opaque sprite under the cursor.[1]
the timeoutMouse	Determines whether mouse clicks reset *the timeoutLapsed* property to 0. See Chapter 11.

[1] See "*Which Sprite Was Clicked On or Rolled Over?*" earlier in the chapter.
[2] Refer to Chapter 12, *Text and Fields*, in *Director in a Nutshell* for details on using the *mouseChar*, *mouseItem*, *mouseLine,* and *mouseWord* properties.

Mouse Tasks

Table 9-7 shows the handler(s) and scripts(s) used to accomplish common tasks related to the mouse. The trick is to place the correct type of handler in the correct type of script, so that it will be called when the event of interest occurs. For the purposes of this table, I've made up the following short-hand notations:

- "*Mouse click handlers*" include *on mouseDown*, *on mouseUp*, *on rightMouseUp*, *on rightMouseDown*, and *on mouseUpOutside*.
- "*Mouse rollover handlers*" include *on mouseEnter*, *on mouseLeave*, and *on mouseWithin* (new in D6). In D5 use *the rollover()* function or *the rollover* property instead).
- "*Frame handlers*" include *on exitFrame*, *on enterFrame*, *on idle*, and *on prepareFrame*.

Note that the effect of these handlers depends on the script in which they are placed.

Table 9-7: Common Mouse Tasks

To Detect This Event or System Property:	Use This Type of Handler, Property or Function:	In This Type of Script:
Mouse clicks on all sprites using a particular cast member	Mouse click handler	Cast member script

Table 9-7: Common Mouse Tasks (continued)

To Detect This Event or System Property:	Use This Type of Handler, Property or Function:	In This Type of Script:
Mouse clicks on a sprite	Mouse click handler	Sprite script
Mouse clicks ignored by all sprites in a frame	Mouse click handler	Frame script
Mouse clicks throughout the entire movie otherwise ignored by other scripts	Mouse click handler	Movie script
All left mouse clicks *before* they are passed to individual sprites	*the mouseDownScript* or *the mouseUpScript*	Movie script
Rollovers on all sprites using a particular cast member	Mouse rollover handler	Cast member script
Rollovers for an individual sprite	Mouse rollover handler	Sprite script
Rollovers for all sprites within a frame	`rollover`*(n) or the rollover* within a frame handler	Frame script
Mouse rollovers throughout the entire movie.	`rollover`*(n) or the rollover* within a frame handler	Movie script
Location of last mouse click	*the clickLoc* property in a mouse click handler	Any script
Current cursor position	*the mouseH* and *the mouseV* properties in any handler	Any script
The last sprite clicked	*the clickOn* property in any handler	Any script
A sprite's channel number	*the spriteNum of me* or *the currentSpriteNum* properties in a mouse click handler or mouse rollover handler	Sprite or Cast member script
Wait in a frame until the user clicks the mouse	*the mouseDown* property in an *exitFrame* handler or the Tempo Channel *Wait for Mouse Click* option	Frame script (or Tempo channel)

Creating Draggable Sprites

There are a number of ways to allow the user to drag a sprite around the Stage. You can set *the moveableSprite of sprite* property, or you can use the *Moveable* checkbox in the Sprite Inspector or Sprite Toolbar.

You can also use the following script to make a sprite moveable. (In Director 5, you must manually *puppet* the sprite to move it in this manner. Director 6's auto-puppeting feature handles this for you.) Note that it accounts for the potential offset between the click location and the *regPoint* of the sprite's cast member to prevent the sprite from jumping when it is first clicked. This is also a good

example of using point-based math. Refer to Chapter 5 in *Director in a Nutshell* and Chapter 6 of this book.

Example 9-6: Dragging a Sprite

```
on mouseDown me
  set mySprite = the spriteNum of me
  set offset = the loc of sprite mySprite - the clickLoc
  repeat while the stillDown
    set the loc of sprite mySprite = ¬
        point (the mouseH, the mouseV) + offset
    updateStage
  end repeat
end mouseDown
```

Mouse Traps

There are a number of ways to prevent mouse events from being processed. See "*Mouse-Opaque Sprites"* at the beginning of this chapter for important details. See item 3 below for a way to prevent rollover events from being recognized.

1. Use *the lastClick* to prevent rapid-fire mouse clicks from being acknowledged (see Example 9-5).
2. Send the user to a frame in which no sprites have mouse handlers attached, so that mouse events are ignored.
3. Use an unstroked, unfilled QuickDraw rectangle (a shape sprite with a zero-width border) in the top-most animation channel with a dummy *mouseUp* and *mouseEnter* handlers. You can alter its *visible of sprite* or *loc of sprite* properties to turn it on or off.
4. You can disable or enable mouse events by calling the handler in Example 9-7 with a Boolean flag, such as *mouseTrap(TRUE)* or *mouseTrap(FALSE)*: Note that this only enables/disables events generated by the left mouse button, not the right mouse button.

Example 9-7: Build a Better Mouse Trap

```
on mouseTrap flag
  if (flag = TRUE) then
    -- Disable mouse events
    set the mouseDownScript = "stopEvent"
    set the mouseUpScript   = "stopEvent"
  else
    -- Allow mouse events
    set the mouseDownScript = EMPTY
    set the mouseUpScript   = EMPTY
  end if
end mouseTrap
```

The *Buddy API* Xtra (*http://www.mods.com.au/budapi*) claims to be able to disable mouse events under Windows. (I have not personally verified this claim.)

Simulating Mouse Events

Lingo does not provide a way to send a mouse event to Director's event queue or to change the cursor's position (*the clickLoc*, *the mouseH*, and *the mouseV* are all read-only).

The *DirectOS* Xtra (*http://www.directxtras.com/do_doc.htm*) claims to be able to set the cursor position and generate a mouse click for any one of three buttons under Windows. (I have not personally verified these claims.)

The *Buddy API* Xtra (*http://www.mods.com.au/budapi*) claims to be able to set the cursor position under Windows and also restrict the cursor to certain subsections of the screen. (I have not personally verified these claims.) A Macintosh version of the Xtra should be available by the time you read this, but it may not include all the functionality of the Windows version.

Both the DirectOS and Buddy API Xtras provide many other OS-level functions. See also Chapter 10, *Keyboard Events,* and Chapter 14, *External Files*.

The *SetMouse* Xtra (*http://www.gmatter.com/donationware.html* or *http://www.scirius.com*) sets the position of the cursor (cross-platform).

You can manually send mouse messages using *sendSprite*, *sendAllSprites,* and *call*, such as:

```
sendSprite (spriteNumber, #mouseDown {, args...})
sendAllSprites (#mouseUp {, args...})
```

You can also call a mouse handler in a specific script, using:

```
call (#mouseDown, script n {, args...})
```

You can send a message to a script instance, using:

```
call (#mouseDown, scriptInstance  {, args...})
```

If using Director 4.0.4 or Director 5.0, use the form:

```
mouseDown (script n {, args...})
```

as described in the *Lingo Issues* ReadMe file that came with the 4.0.4 update.

Mouse messages sent via *call* are sent only to the specified entity. They are *not* passed to additional scripts through the usual messaging hierarchy.

Custom Mouse Events

Some Sprite Xtras may generate their own mouse-related events that are more useful than Director's default events. For example, the Custom Button Xtra's sprites generate *buttonClicked* events, but only if the Custom Button is enabled. *MouseUp* and *mouseDown* events are generated even for disabled custom buttons, which is often undesirable. Likewise, version 3.10 (and later) of the Pop-up menu Xtra

generates a *menuItemSelected* event when the user selects a valid item from a pop-up menu sprite.

Flushing Mouse Events

Lingo does not provide a mechanism for flushing pending mouse or keyboard events from Director's event queue. We saw earlier how to prevent double-clicks or create a "mouse trap" to absorb all mouse events.

The *FlushEvents* Xtra (*http://fargo.itp.tsoa.nyu.edu/~gsmith/Xtras)* is available for the Macintosh only (and I haven't tried it personally). The Buddy API Xtra will disable mouse events, although it doesn't specifically flush existing events. In most cases, there should be no need to flush mouse events. You should instead structure your Lingo to allow Director to process mouse events continuously, and simply ignore them.

Mouse Event Idiosyncrasies

Mouse events are very reliable when you structure your Lingo correctly, but there are quirks. Consult this list of idiosyncrasies and work-arounds prior to tearing your Lingo apart and your hair out. Also see the quirk list at *http://www.updatestage.com*.

MIAWs

Sprites on the main Stage may receive rollover events even when they are obscured by a MIAW placed over the Stage. This is especially an issue with rollovers that play sounds. The user will be confused when sounds are triggered for no apparent reason. Avoid using MIAWs on top of sprites with rollover Behaviors or trying to use a cursor-opaque sprite as the background of the MIAW or in the foreground of the main movie (see the earlier section, "Cursor-*Opaque Sprites,"* and the following section, *Shockwave*).

MIAWs will continue to trap mouse events over their *rect of window* area even if the MIAW is closed. You must use *forget window*, or move the window off-screen to prevent it from trapping mouse events. See Chapter 6, *The Stage and MIAWs*, in *Director in a Nutshell*.

Shockwave

Unfortunately, Shockwave does not work identically with every version of every browser on all platforms. This is often beyond Macromedia's control.

Allegedly, MIE 4 on the Macintosh doesn't pass *mouseUp* events to Shockwave movies. Search the 1998 DIRECT-L archives (cited in the *Preface*) for keywords such as "*MIE*" to find one workaround posted by Andrew White.

Shockwave movies also allegedly respond to rollover events even when the browser window is not in the foreground (see the similar problem with MIAWs in the previous section).

Mouse click response in Netscape 4 on the Macintosh may be sluggish through no fault of Macromedia's. Again, search Macromedia's web site or the DIRECT-L archives for details and possible work-arounds.

It has also been reported that Shockwave under Windows may not recognize mouse clicks when the mouse buttons have been configured for left-handed users in the *Mouse* control panel.

Editable Text Fields

The *mouseUp* event is sent to an editable field sprite only if the field does *not* have focus. Once it gains focus, an editable field sprite responds to *mouseDown* events but not *mouseUp* events. The *mouseUp* events are sent straight to the frame script (bypassing the cast script or the scripts of any obscured sprites). If the editable field sprite loses focus, it will again accept a single *mouseUp* event.

Right Mouse Button under Window

Under Windows, Director gets confused when multiple buttons are pressed simultaneously. *The stillDown* turns `FALSE` even if you hold down the original mouse button while clicking another mouse button.

While the right mouse button is held down, any left mouse button clicks generate *rightMouseDown* and *rightMouseUp* events, rather than the usual left mouse events (*mouseDown*, *mouseUp* and *the mouseDownScript* and *the mouseUpScript*).

If the left and right mouse buttons are released simultaneously, subsequent left mouse clicks *continue* to generate right mouse events. The error is not corrected until the actual right mouse button is clicked separately. The same bug occurs in D5.0.1, D6.0.2, and D6.5 and presumably other versions. One solution is to duplicate mirror all *mouseDown* and *mouseUp* handlers with *rightMouseDown* and *rightMouseUp* handlers. An Xtra that disables individual mouse buttons may provide a complete solution.

Menu Bars

In version 4.0.3 and earlier of Director, mouse clicks in the top 20 pixels of the screen would be ignored because that area was reserved for the menu bar, even if no menu was in use. Mouse events could be sent manually using the techniques described above from a *mouseDown* handler in a frame script.

Custom Buttons

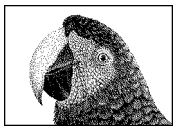
Custom Buttons did not receive *mouseEnter*, *mouseWithin,* and *mouseLeave* events in D6.0. Update to Director 6.0.2 or later. Even so, these events become extremely sluggish if *the idleHandlerPeriod* is not 0.

Rollovers

Don't use *rollover(n)* where `n` is a sprite that is not present in the current Score frame. Director will use the bounding rect of the last sprite to use that sprite

channel to check for rollovers. Use *mouseEnter* and *mouseLeave* events instead, or move the sprites in question off stage in a prior frame.

Mouse events are the core of Director, even more so with new mouse events in D6. Revisit this chapter whenever Director is not executing the Lingo you expect it to. Improper mouse handling is a common culprit.



CHAPTER 10

Keyboard Events

Keyboard Events

Whenever a standard (non-modifier) key is pressed or released, Director generates a *keyDown* or *keyUp* event. Only script attached to editable field sprites, frame scripts, and movie scripts receive keyboard events. All field sprites, even non-editable ones, can receive the standard sprite events (*mouseUp*, *mouseDown*, *mouseEnter*, *mouseLeave*, and so on) Refer to Chapter 7, *Strings,* for details on manipulating strings and text. See especially Example 7-7. Refer to Chapter 12, *Text and Fields,* in *Director in a Nutshell* for more details on working with fields.

Keyboard characters are sent automatically to an editable field sprite with keyboard focus, *unless* the *keyDown* event is intercepted by an *onkeyDown* handler in a sprite or castmember script first (or explicitly stopped in *the keyDownScript* primary event handler).

The key being pressed "rides along" with the *keyDown* event. If an editable field sprite receives the *keyDown* event, the last key pressed will be added to the field. Non-editable fields, editable fields without keyboard focus, and non-field sprites do not receive *keyDown* and *keyUp* events. Rich text sprites can not be edited at run time. You must use field sprites for dynamic text.

Your *keyDown* handler will be called even when special characters, such as the arrow keys, are pressed. Simply *pass* these onto the field sprite.

Fields will automatically recognize the arrow keys and Delete key to perform rudimentary editing (see *"Editing Keys"* later in this chapter).

The Tempo channel can be used to wait for a key press (or mouse click) before advancing the playback head.

Author-Time versus Runtime Keyboarding

Director uses numerous keyboard shortcuts during authoring. Test your keyboard event handling from a Projector. For example, the `Enter` key on the numeric keypad is valid at runtime, but during authoring, it stops the movie. The following types of keys can be tested accurately only from a projector:

- Numeric Keypad Keys
- `F1`, `F2`, `F3`, and `F4` function keys on Macintosh
- Key combinations using `Command` on the Macintosh, or `Control` or `Alt` under Windows, such as for menu shortcuts
- Arrow Keys
- `Page Up`, `Page Down`, `Home`, and `End` Keys

Note that any open Director windows may interfere with keyboard input. When the Paint window has focus, many non-modifier keys switch between various paint tools (See Chapter 13, *Graphics, Color, and Palettes,* covering the Paint window in *Director in a Nutshell.)* To the extent that you do test keyboard events from within the authoring environment, close any windows, especially the Message window.

The Message window accepts keyboard focus and will interfere with *the selStart*, *selEnd*, and *selection* properties.

Standard Keyboard Events

Director generates separate keyDown and keyUp messages when a non-modifier key is pressed and released. Modifier keys (such as Shift and Control) don't generate events by themselves, but rather set system properties such as the shiftDown and the controlDown.

Pressing a non-modifier key also sets the system property *the key Pressed*, representing the current key pressed. Even after a key is released, the system properties *the key* and *the keyCode* contain information about it. A given script may contain handlers that trap both, one, or neither keyboard event (*keyUp* and *keyDown*). If a sprite traps only the *keyDown* event, the *keyUp* event will be passed onto the frame script. The *keyDown* message is sent repeatedly while the key is being held down, as long as the playback head is moving.

Table 10-1 shows how each key event is passed through the Lingo messaging hierarchy. A given event may be handled by a primary event handler, sprite script(s), cast script, frame script, or movie script. Primary event handlers pass keyboard

events by default. Refer also to Example 2-4 and Tables 2-3 and 2-7 in Chapter 2, *Events, Messages, and Scripts*.

Table 10-1: Keyboard Event Passing

Event/Message Type	Message Passing
on keyDown[1]	keyDownScript →Sprite scripts and/or cast script of editable field sprite with keyboard focus[2] →Frame script →Movie scripts.
on keyUp[1]	keyUpScript →Sprite scripts and/or cast script of editable field sprite with keyboard focus →Frame script →Movie scripts.
TAB	Refer to *the autoTab of member* property.
Quit Keys	Sent to OS, unless *the exitLock* is `TRUE`. See Tables 10-4 and 10-5.
Menu Keys[3]	Used with menu bar, if applicable.
OS Keys	Generally not trappable. See Tables 10-4 and 10-5.

[1] A conflict in Director 6 for Windows may prevent Director from receiving keyboard events when RSX is installed.
[2] If the *keyDown* message is *not* intercepted by a sprite or cast script, the typed character is sent to the field itself. Otherwise, it is sent only if the *keyDown* handler includes the *pass* command.
[3] The `Command` key on the Macintosh, and the `Control` and `Alt` keys under Windows may be intercepted by any installed menus with keyboard shortcuts. See the "Menus" section in Chapter 14, *Graphical User Interface Components*, in *Director in a Nutshell*.

Multiple and Repeat Keys

When you hold down a non-modifier key, Director will repeatedly send out the corresponding character. It also repeatedly sends the *keyDown* message. On the Macintosh, *the cpuHogTicks*, not the *Keyboard* control panel, determines the frequency of auto-repeating *keyDown* events. Setting *the cputlogTicks* to 0 generates repeated *keyDown* events most often, but it can interfere with other processes.

If the user presses multiple keys, each one will generate a *keyDown* message when it is pressed and a *keyUp* message when it is released. Therefore, you may receive multiple separate *keyDown* events before receiving any *keyUp* events.

Modifier keys don't generate separate *keyDown* or *keyUp* events, but releasing a modifier key will stop Director from repeatedly sending *keyDown* events even if other keys remain pressed. For example, if you hold down the `Shift` key and the "`A`" key, Director will continually send *keyDown* events. If you then release the `Shift` key, Director will stop generating *keyDown* events even if the "A" key is still depressed. It will then generate the *keyUp* event when the "A" key is released.

Lingo's system properties *the key*, *the keyPressed*, and *the keyCode* always indicate the *last* key *pressed*. Their values from within a *keyUp* handler may not necessarily reflect the last key *released*.

Director stores only the most recent keystroke. You can accumulate keystrokes in a field or accumulate them manually in a variable string or list.

This script will accumulate any keystrokes it traps. Assign it to *the keyDownScript* to track all keystrokes.

Example 10-1: Accumulating and Counting Keystrokes

```
on accumulateKeys
  global gKeyList
  if voidP(gKeyList) then set gKeyList = []
  add (gKeyList, the key)
end accumulateKeys

on startMovie
   set the keyDownScript = "accumulateKeys()"
end startMovie
```

Instead of accumulating the actual keystrokes, you could simply count them:

```
on countKeys
  global gKeyCount
  set gKeyCount = gKeyCount + 1
end countKeys

on startMovie
   set the keyDownScript = "countKeys()"
end startMovie
```

Keyboard Properties

Table 10-2 summarizes system properties regarding the current keyboard state and the last key typed. Note that there is no property to indicate whether a non-modifier key is currently being pressed (although you would ordinarily receive repeated *keyDown* events during this time). That is, there is no *the keyDown* property. Refer to the Xtras cited under "Keyboard Xtras" later in this chapter to detect whether a non-modifier key is currently being pressed.

Table 10-2: Keyboard System Properties and Constants

Property Name	Usage
BACKSPACE	Constant indicating the *Backspace* key at the upper right of the main keyboard (marked "delete" or with an arrow on most keyboards).
the boxType of member	Limits the size of keyboard input fields (possible values are #adjust, #limit, #fixed, and #scroll). See Chapter 12 in *Director in a Nutshell*.
charToNum (the key)	The ASCII value[1] of the last key pressed.
the commandDown	Boolean indicating whether the `Command` key (Mac) or the `Control` key (Windows) is being pressed.

Table 10-2: Keyboard System Properties and Constants (continued)

Property Name	Usage
the controlDown	Boolean indicating whether the `Control` key (either platform) is being pressed.
the emulateMultiButton-Mouse	Boolean indicating whether to treat `Control`-clicks on the Macintosh as equivalent to right mouse clicks on Windows. See Chapter 9, *Mouse Events*.
ENTER	Constant indicating the `Enter` key on the numeric keypad only.
RETURN	Constant indicating the `Return` key on the main keyboard (usually marked "*Enter*" on PC keyboards).
the exitLock	Boolean indicating whether to prevent the user from quitting the Projector with various keyboard shortcuts. Default is `FALSE` (allows user to quit).[2]
the key	The ASCII character[1] of the last key pressed. Does not update in a repeat loop or *keyDown* handler.
the keyCode	The numeric code[1] to the last key pressed, not its ASCII value. Unaffected by modifier keys.
the keyDownScript	Sets primary event handler for *keyDown* events.
the keyPressed	The ASCII character[1] of the last key pressed. Updates in a repeat loop. New in D6.
the keyUpScript	Sets primary event handler for *keyUp* events.
the lastEvent	Time, in ticks, since last key press, mouse click, or any cursor movement.
the lastKey	Time, in ticks, since last key press.
the optionDown	Boolean indicating whether the `Option` key (Macintosh) or `Alt` key (Windows) is being pressed.
the preLoadEventAbort	Determines whether a key event aborts preloading of cast members (see Chapter 9, *Performance and Memory,* in *Director in a Nutshell*).
the shiftDown	Boolean indicating whether either of the two `Shift` keys is being pressed. Not affected by the state of the `Caps Lock` key. See the *CapsLock* Xtra (*http://www.gmatter.com/donationware.html*).
the timeoutKeyDown	Boolean indicating whether keyboard events reset *the timeoutLapsed* property to 0 (default is `TRUE`). See Chapter 11, *Timers and Dates*.

[1] Refer to Appendix A, *ASCII Codes and Key Codes*, for a list of key codes and ASCII values.
[2] See "*Preventing the User from Quitting*" later in this chapter.

Determining Which Key Was Pressed

The best method for deciphering keystrokes depends on the type of key for which you are looking.

Use *the key* property to distinguish printable characters, such as alphanumerics. Use *the keyCode* to distinguish non-printable keys such as the arrow keys. Use *charToNum(the key)* to determine a key's ASCII value.

The key

The key returns the character string of the last key pressed, or even a non-printable character, such as `Ctrl-M`. Multiple keys may set the same value for *the key*. For example, the character "7" may be generated from the "7" key on either the standard keyboard or the numeric keypad.

The keyPressed

The keyPressed returns the same character as *the key*, but its value updates during a repeat loop. *The keyPressed* is *not* a Boolean value indicating whether a key is pressed.

ASCII—charToNum(`character`)

To determine the ASCII value of a character, use:

```
Set asciiVal=chartoNum(the key)
```

This is especially useful for distinguishing between uppercase and lowercase letters. Refer to Appendix C, *Case-Sensitivity, Sort Order, Diacritical Marks, and Space-Sensitivity*. It is also convenient when working with comparing the ASCII values of numeric digits (see Appendix A).

numToChar(`integer`)

NumToChar() converts a number from 0 to 255 into the corresponding character from the ASCII table. Use it to add non-printable characters to an output string, such as:

```
set linefeed = numtochar (10)
```

Note that ASCII values above 127 vary for different fonts. Refer to the character mapping feature of the `FONTMAP.TXT` file as covered in Chapter 12, in *Director in a Nutshell*. Use the Macintosh *Key Caps* desk accessory or the Windows *Character Map* utility (under the Windows 95 Start Menu under `Programs➤Accessories`) to view various characters in different fonts. Under Windows. you can create ASCII characters by holding down the `Alt` key while typing in their ASCII code on the numeric keypad with *Num Lock* on (that is, `Alt`+6+5 will create a capital "A").

The keyCode

The keyCode returns a fixed number based on the key's position on the keyboard. It is unaffected by modifier keys and is unique for each key. Whereas *the key* properties returned by the "7" from the standard keyboard and from the numeric keypad are identical, their *keyCode* properties are different. Use *the keyCode* to detect keys for which there are no Lingo equivalents, such as the arrow keys, or whose ASCII values are not standardized. All keyboards on all platforms seem to send the same key codes for each key.

Refer to Appendix A for a complete list of key codes and ASCII codes.

Detecting the Key's Case

The key differentiates between uppercase and lowercase characters, such as "q" and "Q," but Lingo ignores strings' case when compared using the *equals* sign (=) or the *not equals* sign (<>). The following is therefore somewhat redundant:

```
if the controlDown and (the key = "Q" or the key = "q") then
  statement(s)
end if
```

There is no point in checking the state of the `Shift` key to distinguish between uppercase and lowercase characters, or other keys that have different shifted states. To distinguish between uppercase and lowercase letters, check their ASCII values using *charToNum()*. See Appendix C for complete details, especially Examples C-3 and C-5, which perform case-sensitive comparision and convert text to uppercase. See "*Shift Key*" later in this chapter.

Modifier Keys

Modifier keys are keys that do not generate their own *keyDown* events but rather affect the character generated by other keys. In many cases, they indicate special commands, such as with custom menus (see the "*Menus*" section Chapter 14 of *Director in a Nutshell*.). The modifier keys are different between Macintosh and Windows. Table 10-3 shows the system properties and constants that correspond to various keys.

Table 10-3: Cross-Platform Key Equivalents and Properties

Key	**Macintosh**	**Windows**
Alt Key	N/A	the optionDown
Command Key	the commandDown	N/A
Control Key	the controlDown	the controlDown, or the commandDown
Option Key	the optionDown	N/A
Shift Key	the shiftDown	the shiftDown
Caps Lock	Can't tell without Xtra	Can't tell without Xtra
BACKSPACE	delete	backspace
ENTER[1]	`Enter` key on numeric keypad	`Enter` key on numeric keypad

Table 10-3: Cross-Platform Key Equivalents and Properties (continued)

Key	Macintosh	Windows
RETURN[1]	`Return` key on main Macintosh keyboard	`Enter` key on main PC keyboard

[1] `ENTER` refers only to the `Enter` key on the numeric keypad. The key labeled "*Enter*" on the main portion of PC keyboards generates a `RETURN` character. The `Enter` key on the numeric keypad will stop the movie during authoring, so you can trap it only from a Projector.

The modifier key properties update if the state of the relevant key changes, even during a handler's execution. To ensure that you are checking the *initial* state of a modifier key, store the appropriate Lingo property in a variable.

You can then check the variable throughout the handler without fear of its changing. Note below that we store the value of *the shiftDown*, but not the value of *the key*. The latter does not change during a Lingo handler.

Example 10-2: toring the State of Modifier Keys

```
on keyDown
  set shiftKeyDown = the shiftDown
  if shiftKeyDown and the key = "A" then
    statement
  else if shiftKeyDown and the key = "B" then
    statement
  end if
end
```

In reality, example 10-2 is too short to necessitate storing *the shiftDown*. But suppose you want to detect whether the `Control` key was pressed when the Projector was first started, and if so, skip the introduction of your presentation. You should store *the controlDown* into a global variable (in your *prepareMovie* handler) that you can use long after the `Control` key is released.

`Shift` *Key*

Whereas the `Shift` key capitalizes alphabetic keys, Lingo string comparisons using "=" and "<>" are *case-insensitive*. (See Appendix C for details.) You may want to treat the shifted and unshifted versions of some non-alphabetic keys as the same thing. For example, if the "+" key increases the volume, you may want to check for the unshifted version of the same key ("=" on most keyboards) also. You can also use *the keyCode* to identify a physical key regardless of the `Shift` key's state.

`Caps Lock`

Lingo does not recognize the `Caps Lock` key separately, although using it will capitalize alphabetical characters. See the *CapsLock* Xtra (*http://www.gmatter.com/donationware.html*) to detect its state.

`Alt` *Key (Windows)*

The optionDown reflects the state of the `Alt` key, but `Alt` key combinations are generally trapped by Windows before reaching Director. (There is no *the altDown* property.) By definition, if you are trying to simulate keys that are trapped by Windows, they will be trapped by Windows! `Alt` key combinations such as `Alt-]` are sent to Director, but Alt-A through Alt-Z are not. The `Alt` key will also access custom menus defined with the installMenu command. The Xtras cited under "Keyboard Xtras" below can trap or disable the `Alt` key in limited cases.

`Option` *Key (Macintosh)*

The `Option` key is used for alternate actions, usually in combination with the `Command` key.

Use the optionDown to provide alternate functionality for debugging purposes., as in Examples 3-7, 3-8, and 3-9.

`Command` *Key (Macintosh)*

The `Command` key accesses menu shortcuts on the Macintosh. Certain key combinations are trapped by Director itself during authoring or by the OS during Projector playback and won't reach your handler. Refer to *the exitLock* and *the commandDown*.

`Control` *Key*

The `Control` key, along with *the emulateMultiButtonMouse* property, is used to simulate a multibutton mouse on the Macintosh. Certain key combinations are trapped by Director itself during authoring, or by the OS during Projector playback, and won't reach your handler. Refer to *the exitLock*.

`Use` *the commandDown.* instead of *the controlDown* property when you want the `Control` key to perform an operation under Windows and the `Command` key to perform the analogous operation on the Macintosh (as is the convention).

Special Key Combinations

Many keyboard combinations are intercepted by the OS instead of being sent to your *keyDown* and *keyUp* handlers. These key combinations perform some special OS operation, such as a screen grab, quitting the Projector, or switching to another program. The *exitLock* property disables most of the quit keys, but some key combinations can not be prevented without an Xtra.

Browsers may also intercept certain key combinations, preventing them from reaching your Shockwave movie.

Table 10-4 shows Windows-specific key combinations. To trap the function keys on either platform, use *the keyCode* as described later in this chapter.

Table 10-4: Special Windows Key Combinations

Key Combination	Default Action	How to Prevent It
Escape	Quits Projector.	*the exitLock* = TRUE
Alt-F4	Quits Projector.	*the exitLock* = TRUE
Ctrl-Q	Quits Projector.	*the exitLock* = TRUE
Ctrl-. (period)[1]	Quits Projector.	*the exitLock* = TRUE
Ctrl-W	Does *not* quit Projector, despite claims in documentation.	N/A
Alt Key	Combinations including Alt and an alphabetic key ("A" through "Z") are intercepted by the Windows and not passed to Director.	Not preventable
Alt-Tab	Switches between running Windows tasks.	Buddy API Xtra or DirectOS Xtra[2]
Ctrl-Alt-Del	Brings up task list, with option of restarting computer.	Buddy API Xtra or DirectOS Xtra[2]
Ctrl Key	Executes menu shortcut.	Define shortcuts with *installMenu* command
Ctrl-Print Screen	Screen grab.	Not preventable without Xtra
Windows95 Key	Windows 95 Start Menu.	DirectOS Xtra[2]

[1] The Ctrl-. (period) combination works with the period on both the main keyboard and the numeric keypad.
[2] See "Keyboard Xtras" *later in this chapter*. Some functions may not work under Windows NT.

Table 10-5 shows Macintosh-specific key combinations.

Table 10-5: Special Macintosh Key Combinations

Key Combination	Default Action	How to Prevent It
Escape	No effect.	N/A
Command-Q	Quits Projector.	*the exitLock* = TRUE
Command-. (period)[1]	Quits Projector.	*the exitLock* = TRUE
Command-W	Does *not* quit Projector, despite claims in documentation.	N/A

Table 10-5: Special Macintosh Key Combinations (continued)

Key Combination	Default Action	How to Prevent It
Command-Option-Esc	Gives user option of *Force Quitting* Projector.	Requires third-party Xtra
Restart Key, or Restart Button, or Ctrl-Option-Restart	Gives user option of restarting computer.	Requires third-party Xtra
Command Key	Executes menu shortcut.	Define shortcuts with *installMenu* command
Command-Shift-3	Screen grab.	OSutil Xtra[2]

[1] The Command-, (period). combination works with the period on both the main keyboard and the numeric keypad.
[2] OSutil's *OSSetScreenDump* method can disable basic Macintosh screen captures. See "Keyboard Xtras" *later in this chapter*.

Controlling the Quit Sequence

You may wish to perform some cleanup or post a confirmation or goodbye message when the user quits the projector. To trap special keyboard combinations before they abort the Projector, you'll need to set *the exitLock* to `TRUE`.

Provide an exit button or key combination that quits the Projector or the user will be stuck! Disabling the quit keyboard combinations is impolite for boring presentations.

Example 10-3: Preventing the User from Aborting

```
on prepareMovie --use an on startMovie handler in D5
  set the exitLock = TRUE
end
```

Trapping Quit Keys Manually

If you've set *the exitLock* to `TRUE`, you can then trap the quit keys manually in a movie script. Example 10-4 traps the `Escape` key, `Command-Q`, or `Command-.` (period) on the Macintosh and `Alt-F4`, `Control-Q`, or `Control-.` (period) under Windows. *ConfirmQuit* is a custom routine left as an exercise to the reader. See Chapter 15, *The MUI Dialog Xtra*, or Chapter 14, *Graphical User Interface Components*, in *Director in a Nutshell* for details on creating custom dialog boxes to present the user with multiple choices.

Example 10-4: Trapping the Quit Keys

```
on keyDown
  -- Trap the Escape key by checking for ASCII 27
  -- Also trap Alt-F4 under Windows.
  if charToNum(the key) = 27 or ¬
    (the platform starts "Windows" and the optionDown ¬
```

Example 10-4: Trapping the Quit Keys (continued)

```
     and the keyCode = 118) then
    stopEvent
    if confirmQuit() then
     quit
    end if
    exit
  end if

  if the commandDown then
    -- Prevent Cmd/Ctrl-Q andCmd/ Ctrl-. from aborting
    if (the key = "Q" or the key = ".") then
      stopEvent
      if confirmQuit() then
        quit
      end if
    else
      pass
    end if
  else
    pass
  end if
end keyDown
```

Keyboard Focus

Only editable field sprites are eligible to receive keyboard focus. An editable field receives focus if the user clicks on the field or "tabs" to it. A field will also receive focus if it is the only editable field in the current Score frame. Rich text fields are never editable at runtime. Refer to Chapter 12 in *Director in a Nutshell* for details.

You can pass keyboard focus to a particular field sprite by setting its *editable of sprite* property to `TRUE` (even if it is `TRUE` already). Remove keyboard focus by setting *both* its *editable of sprite* and *editable of member* properties to `FALSE`. Test your Lingo scripts with the Message window closed (it may steal the focus if it is open).

If multiple field sprites are editable, there is no easy way to determine which field currently has focus. You can record a field sprite's number in a global variable whenever it receives a *keyDown* or *mouseDown* event, but no events are sent explicitly when a field sprite gains or loses focus nor is there a system property that indicates the field with keyboard focus.

Creating Editable Fields

The *editable of member* and *editable of sprite* properties determine whether the user can edit the field cast member associated with a field sprite at runtime. Prior to D5 field cast members were callled #text cast members.) The member property corresponds to the *Editable* checkbox in the Field Cast Member Properties dialog.

The sprite property corresponds to the *Editable* checkbox in the Sprite Toolbar or Sprite Inspector. (In Director 5, the *Editable* checkbox is to the left of the sprite channels in the Score window.)

A field is user-editable if *either* its *editable of sprite* or *editable of member* property is `TRUE`. Non-field sprites are never editable, even if their *editable of sprite* property is `TRUE`.

Auto-Tabbing

Director allows you to tab between editable fields by setting *the autoTab of member* property to `TRUE` or setting the *Tab to Next Field* checkbox in the Field Cast member Info Dialog box.

AutoTab determines whether the `Tab` key is intercepted by an editable field or whether it passes focus to the next editable field sprite. The sprites' channel numbers, not their on-stage positions, determine the tabbing order. *AutoTab* does not determine whether a field sprite can *receive* focus. That is determined by whether the field is editable.

You must manually trap the `Return` key and arrow keys to use them to navigate between editable fields. See Example 10-10.

Highlighted Text and the Insertion Point

Use the *hilite* command or *the selStar*t and *the selEnd* properties to highlight a portion of a field, and *the selection*, *the selStart*, and *the selEnd* properties to determine the currently highlighted characters. The Message window interferes with *the selStart*, *selEnd*, and *selection* properties. Activate your test scripts by attaching them to buttons rather than testing them from the Message window.

You must set *the selEnd before* setting *the selStart* to get them to update reliably.

The selStart and *the selEnd* represent positions *after* characters. If *the selStart* and *the selEnd* are both 1, the insertion point is between characters 1 and 2. Set them both to 0 to insert keystrokes before the first character.

Example 10-5: Setting the Insertion Point

```
on setInsertionPoint fieldSprite, cursorPosition
  -- Force focus onto this sprite
  set the editable of sprite fieldSprite = TRUE
  -- Set the selEnd first for reliability
  set the selEnd   = cursorPosition
  set the selStart = cursorPosition
```

Example 10-5: Setting the Insertion Point (continued)

```
  return cursorPosition
end setInsertionPoint
```

This forces focus onto a sprite and positions the cursor at the end of its field:

```
    on beginSprite me
      set mySprite = the spriteNum of me
      setInsertionPoint (mySprite, ¬
           length(field (the member of sprite mySprite)))
    end beginSprite
```

Setting *the selEnd* and *the selStart* to different values highlights the text between them (or you can use the *hilite* command).

Example 10-6: Setting the Highlight in a Field

```
on hilightText fieldSprite, startChar, endChar
  set the editable of sprite fieldSprite = TRUE
  -- Set the selEnd first for reliability
  set the selEnd   = endChar
  set the selStart = startChar-1
  return the selection
end hilightText
```

This highlights characters 1 to 3 of the current sprite (assumed to be a field):

```
    hilightText (the currentSpriteNum, 1, 3)
```

Filtering Keyboard Input

In the default case, any characters typed by the user will appear in an editable field. You can intercept keyboard input, *before* it appears in the field, with a *keyDown* handler attached to the field sprite. Movie-wide key events can also be trapped with *the keyDownScript*. Refer to the previous examples.

Keyboard events not trapped by a field's sprite or cast script are passed to the frame script or movie scripts. If you attach a *keyDown* handler to a sprite, you must include the *pass* command for the field cast member to actually receive the keyboard character(s).

It is possible to attach multiple *keyDown* handlers to a single sprite. Table 10-6 shows what happens when *keyDown* events are passed or consumed explicitly or implicitly. See Table 10-1 for the event passing order.

Table 10-6: KeyDown Event Passing

Command Used in First Behavior	Passed to Other Behaviors?	Passed to Editable Field?
pass	Yes	Only if all Behaviors issue pass

Table 10-6: KeyDown Event Passing (continued)

Command Used in First Behavior	Passed to Other Behaviors?	Passed to Editable Field?
<None specified> or *dontPassEvent*	Yes	No, but later Behaviors can manually append character to field
stopEvent	No	No

Don't forget to trap undesirable keys when requesting user input. Example 10-7 shows how to filter out certain keys and perform some action when the `Return` or `Enter` key is pressed. This *keyDown* handler attached to a field sprite prevents the user from entering any spaces.

Example 10-7: Disallowing Characters

```
on keyDown me
  if the key = RETURN or the key = ENTER then
    -- Process the user entry
    set contents = the text of member  ¬
      (the member of sprite (the spriteNum of me))
    alert "You entered" && contents
  else if the key = SPACE then
    -- Beep if they enter a SPACE
    beep
  else
    -- This sends the key event onto the editable field
    pass
  end if
end keyDown
```

You generally will not want to allow the user to include the `Return` character in a field. Trap the `Return` key unless allowing multiline inputs.

The previous example traps the `Return` character explicitly. Example 10-8 rejects the `Return` character because it allows only the digits 0 through 9 to be passed through. It assumes that there is a separate *submit* button that reads the field's contents.

Example 10-8: Allowing Only Specific Characters

```
on keyDown
  if ("0123456789" contains the key) then
    pass
  else
    stopEvent
  end if
end keyDown
```

Reader Exercise: Modify Example 10-8 to allow other characters used to enter numbers, such as "=," "-", and ".".

To limit the length of a field, you can set *the boxType of member* to either #fixed or #limit. Refer to Chapter 12 in *Director in a Nutshell*. That's usually easier than trying to track the length of the field manually. If you try to stop user input after a certain number of characters, you must allow for the possibility that the user is pressing the `Delete` key or that highlighted characters will be deleted by the next key pressed. It is easier to check the final length of the string once the user has "submitted" it, rather than testing it at every keystroke.

Simulating a Password Entry Field

Because an *onkeyDown* handler in a score script or castscript intercepts keys before they are passed to the field, you can modify them before they are displayed. You could capitalize all letters, for example. Example 10-9 simulates a password entry field using asterisks. Note that the field is updated manually, and that the original key is *not* passed with *pass* (unless it is one of the standard editing keys). Manually passing characters to a field requires us to manually simulate some of the things that Director does automatically if we just pass the characters (we want to send asterisks instead). Note the tricks we use to figure out whether to insert the characters or replace highlighted text and how we reset the cursor insertion point.

Example 10-9: Password Entry Field

```
property pPassword
property pMyMember

on beginSprite me
  set pMyMember = the member of sprite (the spriteNum of me)
  -- Clear the password field
  set pPassword = EMPTY
  put EMPTY into member pMyMember
end

on keyDown me
  -- Mask the field with asterisks.  Change this to
  -- "set maskCar = the key" to see password entry
  set maskChar = "*"

  if the key = RETURN or the key = ENTER then
   -- Check the password when they hit RETURN or ENTER
   -- Perhaps check name against a master list or file.
    if verifyPassword (pPassword) = TRUE then
      go frame "Top Secret"
    else
      alert "Access Denied"
    end if
  else if isEditingKey() then
   -- Allow the editing keys (arrows, delete) to pass
    pass
  else
```

Example 10-9: Password Entry Field (continued)

```
    -- Determine what portion of the field to replace
    set insertPoint = max(1, the selStart + 1)
    set endPoint    = max(1, the selEnd)
    -- We're inserting at the cursor location
    if insertPoint = endPoint then
       -- Add the key to the secret password
      put the key before char insertPoint of pPassword
       -- But display the masking character (asterisks)
      put maskChar before char insertPoint [LC]
          of member pMyMember
    else
      -- Replace highlighted characters similar to above
      put the key into char insertPoint to endPoint [LC]
          of pPassword
      put maskChar into char insertPoint to endPoint [LC]
          of member pMyMember
    end if
    -- Update the cursor insertion point manually
    -- Set the selEnd first for reliability
    set the selEnd   = insertPoint
    set the selStart = insertPoint
  end if
end keyDown

-- Check for common editing keys that we'll allow through (See Example 10-10.)
on isEditingKey
  case (the key) of
    BACKSPACE, TAB:  return TRUE
  end case

  case (the keyCode) of
    -- This checks for delete key and the arrow keys
    51, 117, 123, 124, 125, 126:
      return TRUE
    otherwise:
      return FALSE
  end case
end isEditingKey

-- Write your own verification routine
on verifyPassword password
 If password = "platypus" then
 else return TRUE
 end if return FALSE
end
```

Editing Keys

By default, editable fields support only the `Delete` (`Backspace`) key and the arrow keys for editing. The left and right arrow keys move the cursor one character, and the up and down arrow keys jump to the beginning and end of the field. The `Tab` key jumps between fields (assuming *the autoTab of member* property is `TRUE`.).

You must manually implement cut, copy, paste. You may want to make the up and down arrow keys move between fields. Use the Return key to jump to the next field of a multifield input screen, or submit a single field entry.

Table 10-7 shows *the keyCodes* of common editing keys you can use to control keyboard navigation between multiple editable fields.

Table 10-7: Editing Keys

Key	ASCII	the keyCode
RETURN	13	36
Left Arrow	28	123
Right Arrow	29	124
Up Arrow	30	126
Down Arrow	31	125
Help (or insert)	5	114
Home	1	115
Page Up	11	116
Page Down	12	121
End	4	119
BACKSPACE	8	51
Del (keypad)	127	117
ENTER	3	76

You can allow the user to navigate between editable fields (see *"Keyboard Focus"* earlier in this chapter) using the up and down arrows, and the `Return`, `Home`, and `End` keys, as follows. (Use *the autoTab of member* property or *AutoTab* checkbox to jump between fields without Lingo scripting.)

Example 10-10: Special Handling of Editing Keys

```
on keyDown
 case (the keyCode) of
    125: -- Down arrow
          -- Perhaps send focus onto the next field
          -- by setting its editable of sprite property
          126: -- Up arrow
          -- Perhaps send focus onto the previous field
          115: -- Home key
          -- Send keyboard focus to first field
          119: -- End key
          -- Switch keyboard focus to last field
    otherwise:
```

Example 10-10: Special Handling of Editing Keys (continued)

```
        if the key = RETURN then
           -- Process field or jump to next field
        else
          pass
        end if
  end case
end
```

Numeric Keypad Input

Director ignores input from the numeric keypad by default. To allow numeric keypad input, you must evaluate *the keyCode* generated by each key, and if it corresponds to a key from the numeric keypad, append the appropriate character to the field.

Table 10-8 shows the keys you may want to trap for interpreting numeric keypad input. Note that *the keyCode* numbers skip a beat between the "7" and the "8" on the keypad. In Director 5, the "7" and "8" keys on the keypad erroneously returned the same code

Whenever dealing with numbers, don't confuse the characters "0" through "9," whose ASCII values range from 48 to 57, with the ASCII values 0 through 9, which are all unprintable control characters. See Appendix A.

Example 10-11 reads numbers from the keypad (note the workaround to handle the fact that the *keyCodes* skip number 90).

Example 10-11: Trapping the Numeric Keypad Keys

```
on keyDown me
  -- This handles the keypad chars 0 through 7
  if (the keyCode >= 82) and (the keyCode <= 89) then
    set thisChar = numToChar (the keyCode-34)
  -- This handles the keypad chars 8 and 9
  else if (the keyCode >= 91) and (the keyCode <= 92) then
    set thisChar = numToChar (the keyCode-35)
  else
    -- Let all other keys pass through
    set thisChar = the key
  end if
  put thisChar after member ¬
        (the member of sprite (the spriteNum of me))
end
```

Table 10-8: Numeric Keypad Codes

Key	ASCII	keyCode
. (period)	46	65
Enter	3	76
+	43	69
-	45	78
*	42	67
/	47	75
=	61	81
num lock/clear	27	71

Key	ASCII	keyCode
0	48	82
1	49	83
2	50	84
3	51	85
4	52	86
5	53	87
6	54	88
7	55	89
8	56	91
9	57	92

Function Keys

To trap the function keys, use the key codes shown in Table 10-9. Like other key codes, these are the same on the Macintosh and Windows.

Table 10-9: Function Keys

Function Key	ASCII	keyCode
F1	16	122
F2	16	120
F3	16	99
F4	16	118
F5	16	96
F6	16	97
F7	16	98
F8	16	100
F9	16	101
F10	16	109
F11	16	103
F12	16	111
F13	16	105

Table 10-9: Function Keys (continued)

Function Key	ASCII	keyCode
F14	16	107
F15	16	113

Keyboard Tasks

Table 10-10 shows the handler(s) and scripts(s) used to accomplish common tasks related to the keyboard. The trick is to place the correct type of handler in the correct type of script, so that it will be called when the event of interest occurs.

Table 10-10: Common Keyboard Tasks

To Detect	Use This Type of Handler or Property	In This Type of Script
Keyboard input for all sprites using a particular cast member	*on keyDown*	Castmember script
Keyboard input for a sprite	*on keyDown*	Sprite script
Keyboard input ignored by all sprites in a frame	*on keyDown*	Frame script
Keyboard input throughout the entire movie otherwise ignored by other scripts	*on keyDown*	Movie script
All keyboard input *before* it is passed to individual sprites or frames.	set *the keyDownScript*	Place primary event handler in movie script
Keyboard input from numeric keypad	*on keyDown*	Sprite or cast member script
Key being pressed at any time	set *the keyDownScript*	Place primary event handler in movie script
Key being released at any time	set *the keyUpScript*	Place primary event handler in movie script
Modifier keys	Check the *commandDown*, *controlDown*, *optionDown*, or *shiftDown* property	Any
The last key pressed	Check the *key*, *keyPressed*, *keyCode* property	Any
Whether a specific key is pressed	Use third-party Xtras described under “keyboard xtras” later in this chapter	Any

See “Keyboard Focus” earlier in the chapter to force keyboard focus on a sprite. There is no easy way to detect which sprite has keyboard focus.

Key Trap

There are a number of ways to prevent key events from being processed. You can simply send the user to a frame in which no sprites have key handlers attached.

You can disable or enable key events by calling the following handler with a Boolean flag, such as *keyTrap(TRUE)* or *keyTrap(FALSE)*.

Example 10-12: Key Trap

```
on keyTrap flag
  if (flag = TRUE) then
    -- Disable key events
    set the keyDownScript = "stopEvent"
    set the keyUpScript   = "stopEvent"
  else
    -- Allow key events
    set the keyDownScript = EMPTY
    set the keyUpScript   = EMPTY
  end if
end keyTrap
```

Simulating Keyboard Events

Lingo does not provide a mechanism for setting *the key*, *the keyCode*, or *the keyPressed* or for directly sending a keyboard event to Director’s event queue. (See the “keyboard Xtras” later in this chapter.)

You can manually send keyboard messages using *sendSprite*, *sendAllSprites*, and *call*, such as:

```
sendSprite (spriteNumber, #keyDown {, args...})
sendAllSprites (#keyUp {, args...})
```

You can also call a *keyDown* handler in a specific script, using:

```
call (#keyDown, script n {, args...})
```

You can send a message to a script instance, using:

```
call (#keyDown, scriptInstance {, args...})
```

If using Director 4.0.4 or Director 5.0, you can use the forms:

```
keyDown (script n {, args...})
send (#keyDown, scriptInstance {, args...})
```

Keyboard messages sent manually are sent only to the specified entity. They are *not* passed to additional scripts through the usual messaging hierarchy.

Keyboard Xtras

Xtras provide a number of keyboard-related functions not possible via Lingo alone.

Flushing Keyboard Events

Lingo does not provide a mechanism for flushing pending mouse or keyboard events from Director's event queue. A number of older (often unsupported) XObjects, such as "*Johnny*" and "*MISC_X*" flushed pending mouse and keyboards events. The *FlushEvents* Xtra (*http://fargo.itp.tsoa.nyu.edu/~gsmith/Xtras)* by Geoff Smith is available for the Macintosh only (and I haven't tried it personally).

The OSutil Xtra (*http://www.ddce.cqu.edu.au/imu/tools/Director/Xtras*) by Paul Farry has an *OSFlushEvents* method that purportedly flushes mouse and keyboard events. I've used this Xtra for other chores with good success, but I have no experience with this method.

The KeyPoll Xtra (see next section) can disable keyboard input but doesn't flush pending events.

Polling for Keys and Multiple Simultaneous Keys

The KeyPoll Xtra (*http://www.gmatter.com/donationware.html*) (developed by Brian Gray, formerly of Macromedia) allows you to do the following on all major platforms:

- Check whether the key with a specific *keyCode* is being pressed.
- Get a list of the *keyCodes* of all keys currently being pressed.
- Disable/enable all keyboard events from reaching the system-level queue.

Allegedly, the Macintosh reports up to two character keys being pressed in addition to any combination of the five modifier keys or the arrow keys. Brian Gray reports that more than two characters keys are sometimes detectable. Details can be found in *Inside Macintosh: Macintosh Toolbox Essentials* at *http://developer.apple.com/*.

The *CapsLock* Xtra (also at *http://www.gmatter.com/donationware.html*) can detect the state of the `Caps Lock` key.

See also the Buddy API and DirectOS Xtras in the next section.

Sending Key Events and Disabling Keys

The *Buddy API* Xtra (*http://www.mods.com.au/budapi*) claims to be able to check which keys have been pressed (and also simulate key events and disable keyboard input) under Windows using the *baSendKeys*, *baKeyIsDown*, *baKeyBeenPressed*, *baDisableKeys*, and *baDisableSwitching* methods. (I have not personally verified these claims, but I have heard uniformly positive comments about Buddy API in general and their tech support was responsive.) A Macintosh version of Buddy API should be available by the time you read this.

The *DirectOS* Xtra (*http://www.directxtras.com/do_doc.htm*) claims to be able check which keys have been pressed (and also disable certain key combinations) under Windows. (I have no personal experience with this Xtra).

Both the DirectOS and Buddy API Xtra provide many other OS-level functions. See also Chapter 9 and Chapter 14, *External Files*.

See also the other Xtras listed earlier.

Keyboard Event Idiosyncrasies

Authoring Caveats

During authoring, Director may intercept certain keyboard events. Refer to the discussion at the beginning of this chapter. When trying to edit or highlight text in editable field sprites, Director's Message window is a frequent trouble-maker. Close all Director windows, or test from a Projector if necessary.

Kiosks Without Keyboards

If you are creating a kiosk without a keyboard, you can use a simulated graphic touch-screen to allow for user input. Simply have each sprite's *mouseUp* handler append the appropriate character to the input field.

RSX Conflict

RSX is a sound driver from Intel that has some conflicts with Director for Windows. RSX can prevent Director for Windows (D6.0, D6.0.1, and D6.0.2) Projectors played full-screen from receiving keyboard events. Macromedia has apparently acknowledged the problem, but no fix has been released as of May 1998. If possible, ask your users to disable RSX.

Keyboard Events to Frame and Movie Scripts

There have been reports that frame scripts and movie scripts may not receive keyboard events unless at least one editable field sprite is present in the current frame. Place one off-stage, if necessary. I have not seen this behavior myself, and it is not clear that it is a separate issue from the RSX conflict.

MIAWs

A movie-in-a-window whose window has focus will intercept keyboard events even if it does not have any editable text fields. You can click on the Stage to ensure that it has focus so that the main movie receives keyboard events properly.

In a movie script of the MIAW, use the code shown in Example 10-13 to send keyboards events to the Stage.

Example 10-13: Passing Key Events from MIAWs

```
on keyDown
  tell the stage
    do (the keyDownScript)
```

Example 10-13: Passing Key Events from MIAWs (continued)

```
  end tell
end keyDown
```

In the main movie, you can define *the keyDownScript* to execute the handler of your choice. The main movie can check *the key* to determine the last key pressed, or even use *moveToFront the stage* to bring itself to the foreground.

In the main movie you might use:

```
on startMovie
  set the keyDownScript = "customKeyDownHandler"
end

on customKeyDownHandler
  put "The key is" && the key
  -- Do any custom keyboard handling here
end
```

Menu Bars

When using a custom menu, Windows intercepts keyboard combinations using the `Alt` and `Ctrl` keys, and the Mac OS intercepts keyboard combinations using the `Command` key.

MouseUp Events

The *mouseUp* event is only sent to an editable field sprite if the field does *not* have focus. See Chapter 9 for details.

Shockwave

Shockwave movies may not receive keyboard events until receiving focus via a mouse click. Some developers add a "Click here to start" button to solve the problem.

The browser may trap certain keyboard shortcuts, preventing those key combinations from reaching your Lingo scripts. Avoid browser-specific keyboard shortcuts.

KeyCode Tester

You'll want your own keyboard tester to check the return values of idiosyncratic keys not listed in this chapter or Appendix A (such as those with higher ASCII values created using the `Option` or `Alt` keys). Attach this script to an editable field sprite. It prints out the results in the Message window.

Example 10-14: Your Own KeyCode Tester

```
on keyDown
  put "key:" && the key
  put "keyCode:" && the keyCode
  pass
end
```

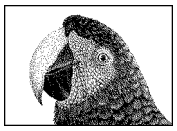
You may need to close the Message window before pressing keys, then open it afterward to see the results. You could get fancy and send *the keyCode* output to another field shown elsewhere on stage. Refer to the "*Keyboard Lingo"* Show Me in the online Help (or Macromedia's Web site), or download the keyboard tester from http://www.zeusprod.com/nutshell/examples.html.

Keyboard Potential

Don't overlook the potential to use your keyboard for good. Try implementing the following:

- Use the keyboard to set the volume (Hint: Use *the soundLevel*).
- Use keyboard shortcuts for custom menu options (see the "*Menus"* section of Chapter 14 in *Director in a Nutshell*).
- You can accept simultaneous input from multiple daisy-chained Macintosh ADB keyboards. Lingo can't tell which keyboard sent the event, so it is fun for collaboration, not gaming.

Keyboard handling in Director is far from elegant, but this chapter has armed you with the knowledge to steer clear of many pitfalls. Refer frequently to the quirk list at *http://www.updatestage.com/* because many keyboard idiosyncrasies encountered are not the fault of your Lingo code.



CHAPTER 12

Behaviors and Parent Scripts

The term *Behavior* is used loosely to describe several different types of Lingo scripts. For now, think of Behaviors as score scripts that are easy to customize, even by novices. Macromedia provides a slew of Behaviors under `Xtras➤Behavior Library`. Refer to the *Behaviors* Show Me demo movie in the online Help to learn the basics of applying existing Behaviors. See Chapter 5, *Creating Interactivity,* in Macromedia's *Using Director* manual for an overview of the available Behaviors. Macromedia also provides numerous *widgets* (premade UI components) and buttons that use Behaviors to create sprites with complex functionality. See `Xtras➤Widget Wizard➤Widget Wizard` and `Xtras➤Widget Wizard➤Button Library`. Refer to Chapter 14, *Graphical User Interface Components,* in *Director in a Nutshell* for a detailed example of using premade widgets and Behaviors.

The *Lingo Behavior Database* is a collection of Behaviors maintained by Renfield Kuroda at *http://www.behaviors.com/lbd/*.

Search the Macromedia site for Technote 08140 for possible ways to distribute protected Behaviors. Refer to the downloadable examples (*http://www.zeusprod.com/nutshell/examples.html*) for a detailed Behavior that simulates the Tempo channel for use in Shockwave (which doesn't obey the Tempo settings).

What Is a Behavior?

The simplest (non-configurable) Behaviors are exactly like Director 5-style score scripts attached to either a sprite or the script channel. To add a "beep-when-clicked" Behavior to a sprite, you can use the simple script.

Example 12-1: A Ludicrously Simple Behavior

```
on mouseDown
  beep
end
```

Even such plain vanilla Director 5-style score scripts appear in the Behavior Inspector. For the remainder of this chapter the terms *score script* and *Behavior* will be used interchangeably.

Director 6 allows each sprite to have multiple Behaviors attached. Although only one Behavior can be attached to each frame in the script channel, you can attach an *on exitFrame* handler to a *sprite* channel to emulate multiple frame scripts.

Support for multiple scripts per sprite allows you to *modularize* your scripts into smaller pieces and attach more scripts as needed. For example, suppose two sprites have the same *mouseEnter* response but different *mouseDown* responses. You could attach the same *on mouseEnter* Behavior to both sprites and then add separate *on mouseDown* Behaviors to each. If their *mouseDown* responses are similar (perhaps differing only in the sound to be played on *mouseDown*), a single customizable Behavior could be written to handle *mouseDown* events for both sprites.

A Behavior can be thought of as a score script with easily customizable attributes, such as which sound to play when an event occurs. They can also be thought of as Parent scripts attached to sprites. Behaviors are instantiated when the playback head enters the sprite span to which they are attached. The custom properties for each instance are stored with the Score data and read back at runtime.

A Behavior's properties persist for the life of the sprite, and Behaviors can access any property of any other Behavior attached to the same sprite using:

```
the property of sprite (the spriteNum of me)
```

Creating Simple Behaviors

The Behavior Inspector can be used as a simple Behavior *Constructor*. Refer again to Chapter 5 in Macromedia's *Using Director* and the *Behaviors* Show Me demo movie.

Example 12-2: Writing Simple Scripts Via the Behavior Inspector

To watch Director construct the script as you select *Events* and *Actions* to add:

1. Open the Behavior Inspector using the "`*`" key on the numeric keypad or `Window➤Inspectors➤Behavior`.
2. Click the *Edit Pane Expander* (see Figure 12-1) to expand the editing pane.
3. Select *New Behavior* from the *Behavior* popup, and name the Behavior.

Example 12-2: Writing Simple Scripts Via the Behavior Inspector (continued)

4. Open the Script window using the *Script* button at the top of the Inspector.
5. Use the *Events* popup and *Actions* popup to build your script.

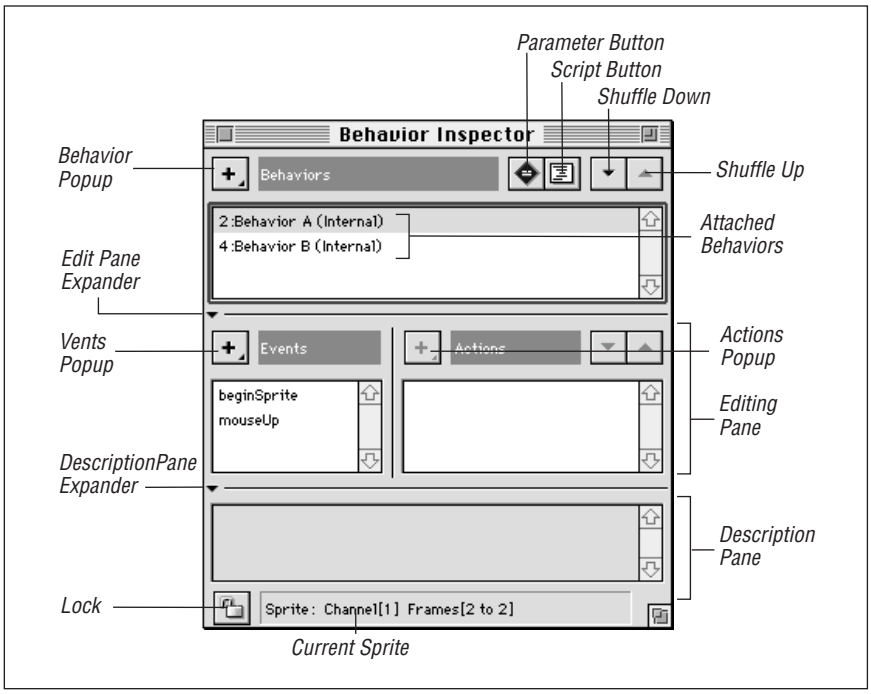


Figure 12-1: Behavior Inspector window

The Behavior Inspector doesn't create true "*Behaviorized*" scripts with all the fancy attributes of Behaviors. It just helps novices to create basic score scripts without typing in the Script window. A true Director 6-savvy Behavior usually has user-defined properties and a minimal help string to explain its use.

Roy Pardi offers a *Behavior Writer Xtra* to ease *some* of the mechanics of writing your own Behaviors. See *http://www.tiac.net/users/rpardi/behaviorwriter/*.

Let's create a Behaviorized version of the beeping button script shown earlier. When the Behavior—shown in Example 12-3 is attached to a sprite Director uses the property list returned by *ongetPropertyDescriptionList()* to create a dialog (see Figure 12-2) that lets the user customize the Behavior. From the dialog, the developer can choose which type of mouse event triggers the beep.

Example 12-3: A simplified beeping Behavior

```
property whichEvent
on mouseUp me
  if whichEvent = #mouseUp    then beep
end

on mouseDown me
  if whichEvent = #mouseDown  then beep
end

on mouseEnter me
  if whichEvent = #mouseEnter then beep
end

on getPropertyDescriptionList
  return [#whichEvent: [#comment: "Initializing Event:", ¬
           #format:  #symbol, ¬
           #range:  [#mouseUp, #mouseDown, #mouseEnter], ¬
           #default: #MouseUp]]
end
```

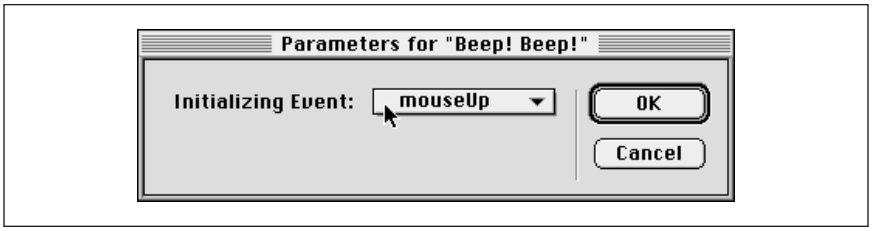


Figure 12-2: Behavior parameter dialog

Even this simplified Behavior is much more complicated than a standard sprite script, and it still just creates a beep! (Macromedia's Behavior Library includes an even more complicated *Sound Beep* Behavior).

In this case, the dialog includes a pop-up menu with three possible values for the `whichEvent` property (`#mouseUp`, `#mouseDown`, and `#mouseEnter`), as defined in the *#range* attribute of the *#whichEvent* property. At runtime, the Behavior beeps if the mouse event (such as *mouseDown*) matches the trigger event chosen for `whichEvent`.

This may seem a poor candidate for a Behavior because it complicates a very simple script. Behaviors, however, allow a novice to add *customized* behaviors without Lingo scripting.

Behaviors are sometimes hard to create and ugly to read, but easy to use.

Using the Behavior Inspector

Apply existing Behaviors by dragging them from the Cast window to a sprite or the script channel, or by using the *Behavior Script* popup in the Sprite Toolbar or Sprite Inspector. A Behavior's *scriptType of member* property must be *#score*, not *#movie* or *#parent*, or it will not appear in Behavior Inspector. If you open the Behavior Library, its scripts show up in the Behavior popup, too.

Open the Behavior Inspector using the "`*`" key on the numeric keypad or `Window➤Inspectors➤Behavior`.

If `View➤Sprite Overlay➤Show Info` is active, you can open the Behavior Inspector using the little green icon that appears next to the selected sprite on the Stage.

You can add your own Behavior Libraries to the ones under the `Xtras` menu, as described in Chapter 4, *CastLibs, Cast Members, and Sprites,* in *Director in a Nutshell* (add "*Library*" to the cast name and drop it in the Xtras folder).

If you are a novice using other people's Behaviors, set `File➤Preferences➤Editors➤Behaviors` to edit scripts in the Behavior Inspector, the Script window. *See* "*Where the Hell Are My Scripts?*" and Table 2-2 in Chapter 2, *Events, Messages, and Scripts*.

Inside the Behavior Inspector

The Behavior Inspector is very malleable. Use the *Edit Pane Expander* and *Description Pane Expander* buttons (see Figure 12-1) to customize it to your liking.

The *Lock Selection* button prevents the Behavior list from changing if the Score selection changes.

The repertoire of *Actions* for automatic script construction is very limited, but it gives you a basic feel for Lingo scripting. The *Wait until Click or Key Press* Action creates incorrect Lingo code (*puppetTempo -8*), and should not be used.

When a Behavior is applied to a sprite or frame, you will be prompted to customize its properties, if applicable.

To change its properties *after* a Behavior has been attached to the Score, use the *Parameter* button in the Behavior Inspector.

To use the *Parameter* Button:

- Highight a sprite or frame with a Behavior attached, then highlight the desired Behavior in the Behavior Inspector.

- At least one property must be declared with the *property* keyword at the top of a Behavior script, or the *Parameters* button will be inactive.
- A Behavior must declare an *on getPropertyDescriptionList* handler, or the *Parameters* button will have no effect.

Behavior Inspector Pitfalls

Selecting the frame script channel and then creating a new Behavior using the Behavior Inspector will create a script in the cast, but it will not appear in the Score until dragged there. (If you highlight a sprite and create a Behavior, it will be attached automatically.)

You can add Behaviors to multiple selected sprites. If multiple sprites are selected, deleting a Behavior via the Behavior inspector deletes it from only the *first* sprite selected. Choose *Clear Script* from the script popup in the Sprite Toolbar or Sprite Inspector to clear all scripts from multiple sprites.

Behaviors and their properties persist only for the life of the sprite to which they are attached. Use global variables or parent scripts for more persistent data.

Objects of Mystery

Now that some of the mystery has been dispelled about Behaviors, let's discuss their cousins, parent scripts. I'll show you why, when, and how to use *Object-Oriented Programming* (OOP). Once you are introduced to the concepts, I'll then cover the terminology in more detail (refer also to the *Glossary)*. Finally, I'll cover some practical examples. When you finish this chapter you'll realize that the "Great and Powerful Oz" is just some guy behind a curtain. So take a deep breath, repeat after me ("Parent scripts, child objects and Behaviors, oh my!"), and soon you'll be more at home with OOP than Dorothy was in Kansas.

For a comparison of object-oriented Lingo with C++, refer to Table 4-1 in Chapter 4, *Lingo Internals,* and see the downloadable Chapter 20, *Lingo for C Programmers*. Read Chapter 12, *Parent Scripts and Child Objects,* in Macromedia's *Learning Director* manual for another perspective on object-oriented programming. Refer also to the *Simple Child Object* and *Multiple Child Objects* Show Me demo movies in the online Help.

A Procedural Stopwatch

Typical Lingo scripting is *procedural* because you create *procedures* (*functions* or *handlers*) to perform a particular task. For example, the *average()* function might average two numbers.

Example 12-4: A Trivial Procedural Example

```
on average a, b
   return (a+b)/2.0
end
```

If a script calls a procedure, the only communication between them is via the value returned by the function to caller.

A function is like a one-night stand. It has a fleeting existence and typically performs a single operation.

Let's create a stopwatch using a procedural approach. The code in Example 12-5 belongs in a movie script. We must use global variables to communicate between the various functions and to maintain the current state of the timer. (This example is *heavily* simplified and not very robust. See Example 11-9 for a robust object-oriented version of this script.) Note that we used the name *runTimer* instead of *startTimer* to prevent conflicts with the Lingo *StartTimer* command.

Example 12-5: A Procedural Stopwatch

```
global gCurrentTime, gStartTime
-- Reset the timer to 0
on resetTimer
  set gCurrentTime = 0
end resetTimer

-- Start the timer running
on runTimer
  set gStartTime = the ticks
end runTimer

-- Stop the timer (assumes timer was running)
on stopTimer
  set gCurrentTime = (the ticks - gStartTime)
end stopTimer

-- Report the timer's value (assumes timer is stopped)
on reportTimer
  return (gCurrentTime/60.0)
end reportTimer
```

We could use our stopwatch to check how long Lingo takes to print out the numbers: from 1 to 100

```
-- Test the timer functions
on testTimer
  resetTimer
  runTimer
  repeat with x = 1 to 100
    put x
  end repeat
  stopTimer
  put reportTimer()
end testTimer
```

This procedural approach is adequate for a single timer, but we would need to create additional global variables to avoid conflicts between multiple timers.

Object-Oriented Programming

Let's dive right in and create an *object-oriented programming* (OOP) version of our stopwatch. OOP is ideal because we can build a timer *object* (a reusable template), and create multiple *instances* (clones) of it that operate independently. Each instance can maintain its own *properties*, which are semi-private variables, as described in Chapter 1, *How Lingo Thinks*.

Lingo allows you to mix procedural and object-oriented programming. Some people go overboard and turn everything into an object. Use whatever is best for a given situation.

An *object* (that is, a copy of a script) can contain several *methods* (handlers) to perform its desired actions. For example, a Timer object might behave like a stopwatch with four buttons (*resetTimer*, *runTimer*, *stopTimer*, and *reportTimer*), each implemented by a different method. An object's methods are stored in a template called a *parent script*. You don't ordinarily use the template directly; you use a copy or *instance* of the template in the form of a child object. This allows you to create multiple independent copies.

You can use parent scripts as semi-private code libraries without the risks of naming conflicts associated with handlers in movie scripts. A handler named *StartTimer* inside an object would *not* conflict with the Lingo *StartTimer* command because they have differing scopes. See "Handler Scope" in Chapter 2.

Although unusual, instead of instantiating a parent script, you can access its handlers as:

```
set variable = someHandler (script "ParentScript", args)
```

You can even access a parent script's properties *without* instantiating it.

The Life and Death of an Object

The terminology required for objected-oriented programming can be summed up in a few sentences. Some of the terminology is redundant or used loosely. (Refer to the *Glossary* for complete definitions of each term used in this chapter.) Click your ruby slippers together as you repeat three times:

A *parent script* defines the *properties* (attributes) and *methods* (functions) of an *object*. A *child object* is an *instance* (copy or clone) of the parent script and is *instantiated* (created) using the *new()* method. An object is disposed of when no variables refer to it.

Let's look at all this in more detail.

Creating a Parent Script

Example 12-6: Creating a Child Object from a Parent Script

To create a child object from a parent script:

1. Create a script cast member to hold the parent script (it is convenient to name the cast member, too). Use the popup in its cast member info window to set its type to *Parent*, or set its *scriptType of member* to *#parent*.
2. Write the Lingo methods for the *parent script* as shown later. You'll need an *on new* method and other optional methods.
3. In a *separate* script (or the Message window) *instantiate* (create an instance of) the parent script using the *new()* function, and store the instance in some variable.
4. Use the instance (the **child** object) created above to call the other methods in the object. Because you specify the child object when calling other methods, Director knows which instance's properties and methods to use.

A Very Simple Parent Script

Here is a very simple parent script. It defines one property and only one method (*crying*) besides the *on new* method.

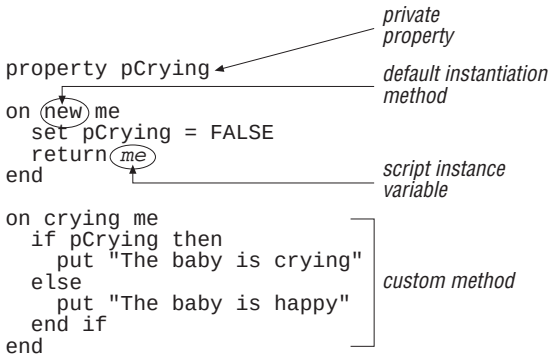


Figure 12-3: Anatomy of a Parent script

If the *on new* handler is omitted, Director uses a default *on new* handler that just returns the child object instance variable (*me*) such as:

```
on new me
  return me
end
```

If the earlier parent script is in a cast member named "Mommy," you can create and manipulate a child object, such as shown in Example 12-7.

Example 12-7: Instantiating and Using a Child Object

```
set baby = new (script "Mommy")
crying(baby)
-- "The baby is happy"
set the pCrying of baby = TRUE
crying(baby)
-- "The baby is crying"
```

You can create as many children as you like, and each can have its own *pCrying* property.

If you don't store the returned *child object instance* (in this case, into the variable *baby*) there is little point in instantiating the object because you won't be able to refer to it later. In fact, it will be disposed of immediately.

Whereas a single child object instance may be stored in a global variable, related child object instances are commonly stored in global lists for later use:

```
global gChildList
if not listP(gChildList) then set gChildList = []
addAt (gChildList, new (script "Mommy"))
```

Using a single global list reduces the number of variables needed. Furthermore each instance can be used to access all the properties of each object.

An Object-Oriented Stopwatch

Here is an object-oriented version of the procedural stopwatch from Example 12-5. This Lingo should be placed in a parent script named "Timer." (See Example 11-9 for a more robust timer object.)

Example 12-8: An Object-Oriented Stopwatch

```
property pCurrentTime, pStartTime

on new me
  return me
end

-- Reset the timer to 0
on resetTimer me
  set pCurrentTime = 0
end resetTimer

-- Start the timer running
on runTimer me
  set pStartTime = the ticks
end runTimer

-- Stop the timer (assumes timer was running)
```

Example 12-8: An Object-Oriented Stopwatch (continued)

```
on stopTimer me
  set pCurrentTime = (the ticks - pStartTime)
end stopTimer

-- Report the timer's value (assumes timer is stopped)
on reportTimer me
  return (pCurrentTime/60.0)
end reportTimer
```

Note these differences from the procedural version in Example 12-5:

- *Property* variables (beginning with the letter "p" for clarity) are used instead of *global* variables. Properties are declared with the keyword *property*, and can contain a different value for each instance of the object (that is, each timer). If we used globals, multiple timers would trample the values held in the globals.
- We added an *on new* method that will be used to instantiate the object.
- The variable `me` indicates the current instance of the child object. It is returned by *new()* when the child object is created and is typically stored in some variable by the caller. It is then used when calling other methods to identify the child object, so that Director can retrieve its properties rather than the properties of some other instance.

We can test the Timer as shown in Example 12-9. It creates two separate instances of the Timer object. The first one times the overall operation, and the second one times the inner repeat loop.

Example 12-9: Testing a Timer Object

```
-- Test the Timer child object
on testTimerObj
  -- Instantiate two timers
  set timer1 = new (script "Timer")
  set timer2 = new (script "Timer")
  -- Start timer1 running
  resetTimer (timer1)
  runTimer (timer1)

  put "Testing the speed of the repeat loop"
  repeat with y = 1 to 10
    -- Start timer2 running
    resetTimer (timer2)
    runTimer (timer2)

    repeat with x = 1 to 1000
      set dummy = 5
    end repeat
    -- Pause and read timer2
    stopTimer (timer2)
    put "1000 Iterations took" && reportTimer(timer2) && "seconds"
  end repeat
```

Example 12-9: Testing a Timer Object (continued)

```
  -- Pause and read timer1
  stopTimer (timer1)
  put "The whole test took" && reportTimer(timer1) && "seconds"
  -- Dispose of timer objects by setting them to zero
  set timer1 = 0
  set timer2 = 0
end testTimerObj
```

Note that we could instantiate dozens of timers without any conflicts or need for multiple global variables. Each timer maintains its own set of properties.

I Gotta Be Me

Director uses the *me* variable to refer to a child object instance within the parent script itself. See also "*Script Instances*" in Chapter 2.

Enter Example 12-10 this into a parent script called "Eden."

Example 12-10: Paradise Lost? Not as Lost as the Reader!

```
property pGender
property pName
property pKnowledge

on new me, gender, name
  set pGender    = gender
  set pName      = name
  set pKnowledge = FALSE
  return me
end

on eatApple me
  set pKnowledge = TRUE
end

on getKnowledge me
  return pKnowledge
end

on ShowInfo me
  put pName && "is" && pGender
end

on GetName me
  return pName
end

on getGender me
  return pGender
end

on testKnowledge me
  if pKnowledge then
```

Example 12-10: Paradise Lost? Not as Lost as the Reader! (continued)

```
    put pName && "is banished"
  else
    put pName && "is innocent"
  end if
end
```

Test it in the Message window. Note that the arguments to the *new()* function call are used to initialize properties for that particular object.

```
set edenList = []
add edenList, (script "Eden", #male,   "Adam")
add edenList, new (script "Eden", #female, "Eve")
```

When calling *ShowInfo()* we specify an object from our *edenList*:

```
showInfo(getAt(edenList,1))
-- "Adam is male"
showInfo(getAt(edenList,2))
-- "Eve is female"
```

When we call *eatApple()*, we again specify a child object instance. We need not be aware what it does internally.

```
eatApple (getAt(edenList,2))

testKnowledge (getAt(edenList,1))
-- Adam is innocent
testKnowledge (getAt(edenList,2))
-- Eve is banished
```

Notice that we indirectly set and accessed the *pKnowledge* property from outside the parent script without even knowing it! The object handles the details for us! One can access any property of an object from *outside* the object, using:

```
put the property of object
```

OOP purists will insist that you should never directly access a property of an object from outside the object. You should instead access them only via accessor methods, such as *getName()* and *getGender()* in Example 12-10.

An accessor method simply returns the value of a property of the object, allowing us to write:

```
getName (getAt(edenList,1))
-- Adam
```

Thus we have not violated the *encapsulation* of the child object. The *getName()* method can be changed without breaking any outside code. Any code that uses the accessor method will continue to work.

Common Errors with Parent-Child Scripting

There are several common errors you will surely commit.

Omitting return `me` *at the end of your on new handler:*
Without `me`, a child object instance will not be returned, and the caller will get a meaningless value.

Omitting `me` *when declaring other methods, such as on eatApple* `me`*:*
Without `me`, you won't be able to set a child object's properties, such as `Knowledge` from within the method.

Not storing the child object instance returned by the new() call:
Unless you store the return value, you won't be able to access the object later (in fact, the instance will be disposed of if nothing references it).

Failing to assign values to the properties:
You can either assign properties in the *new()* method or use a separate *init()* method to allow for reinitialization without reinstantiation.

Trying to access a method within an object without first calling new():
You generally should instantiate the object using *new()* before calling its other methods. *New()* accepts a *parent script* as the first argument. All other methods accept a *child instance* as the first argument, such as:

```
set childObj = new (script "parentScript")
someMethod (childObj)
```

If you decline to instantiate the object first, as described in the tip under "Object-Oriented Programming" earlier in this chapter, you are manipulating the properties of the parent script itself, and not of a child object.

Creating a parent script with type #movie:
Lingo doesn't complain if your parent scripts are of type *#movie*, but the handlers in movie scripts have global scope, which can lead to conflicts or handlers being run unintentionally. If the parent script's type is *#score*, it will inadvertently appear in the script popup in the Sprite Toolbar. (Note that a Behavior's script type should be *#score* for this very reason.)

When to Use Object-Oriented Programming

Parent scripts and child objects have the hallmarks of a long-term relationship. Objects are created manually, tend to shun outsiders, and persist until disposed.

Parent-child scripting is generally a good idea to accomplish the following:

- Create objects that are defined by the current state of their properties, such as timers. An object remembers its own state (i.e., properties) and is therefore much easier to use and maintain without global variables.
- Create objects that persist over time or that require multiple operations. For example, the FileIO Xtra is just an object written in C. It allows you to manipulate external text files (via methods, just like a child object), including the

ability to open, read, write, delete, search, and close the file. The object keeps track of the properties, including the file's name and the location and the last position read within the file.

- Create code that is independent of other code. Objects are *encapsulated* (insulated) so that they can be developed separately from other code. Other Lingo primarily interacts with objects via the defined methods. Although this can simplify development and maintenance of some projects, it is not the panacea some people claim it to be.
- Create classes of objects in a hierarchy. Objects can *inherit* behaviors from *ancestors*, allowing you to create a tree of related objects. For example, a family tree of birds and reptiles may share a common ancestor that has an *egg-Laying* method.

Two objects created from different Behaviors or parent scripts can both have *methods* of the same name that do completely different things.

You can send a single message to multiple objects without knowledge of their internal operation. Each object will respond appropriately, allowing you to deal with differing objects in a uniform way. Refer to "The ActorList" discussed later.

When to Use Behaviors

Behaviors are appropriate when you want a sprite or group of sprites to respond to Director and user events in a certain way. They can be attached to the script channel as well.

Behaviors are like casual dating. A Behavior can be attached to multiple sprite or frames, and a sprite can have multiple Behaviors attached. A Behavior is automatically associated with the sprite to which it is attached, and its lifespan coincides with that of the sprite span.

You should use Behaviors for multiple items with similar properties, such as bouncing balls or space aliens. Although all instances of the object would be intrinsically similar, they can have significantly different values for each property of the object. For example, a ball's properties may include speed, acceleration, diameter, mass, elasticity, and color. Example 8-14 is a sample Behavior that moves a sprite along an elliptical path. The user can specify the foci and major and minor axes of the ellipse.

Child Object References

When you create or use a child object, Director keeps track of how many *references* there are to that object (how many things use it).

An object reference takes the form:

```
<offspring "parentScriptName" referenceCount  IDnumber>
```

If you create an object without storing the return value, the *referenceCount* is just one (the object refers to itself long enough to print in the Message window).

Example 12-11: Child Object References

```
put new (script "parent script")
-- <offspring "parent script" 1 27c252a>
```

Because you made no record of the object, it will be disposed of at Director's whim. When you create an object and store the return value, Director increments the *referenceCount* to 2 because a variable now refers to it as well:

```
set myChild = new (script "parent script")
put myChild
-- <offspring "parent script" 2 27c24f8>
```

Note that a new *IDnumber* was assigned, and this object has no relation to the previously created object.

Now assign a new variable to the existing object:

```
set newVariable = myChild
put myChild
-- <offspring "parent script" 3 27c2548>
```

Note that the *referenceCount* has increased, but the *IDnumber* has stayed the same. When you are done with an object, set any variables that use it to 0. When no variables or lists refer to an object, Director will dispose of it.

```
set myChild = 0
set newVariable = 0
```

There is *no* built-in Lingo to determine the following at runtime:

- A list of objects that currently exist. You need to store the objects as you create them, usually in a list.
- A list of objects derived from a given parent script or the parent script of a given object.
- The number of variables that reference a given object, or what particular variables reference a given object. Short of clearing all global variables, there is no way to clear references to a particular object to ensure that it is released.

You can create a root ancestor for all parent scripts that logs each child object created. You would also explicitly call a *destroy* method to remove the object from the global list.

The unsupported UIHelper Xtra (included with D6.5's *Save as Java* Xtra) has two methods that read Behavior references and their property settings. These are *unsupported*:

getBehaviorMemRef (spriteNum, nthBehaviorNum)
: The function returns the script member number of the *nthBehaviorNum* attached to *spriteNum* in the current frame.

getBehaviorInitializers (spriteNum, nthBehaviorNum)
: The function returns the user-specified values for the properties of the *nthBehaviorNum* attached to *spriteNum* in the current frame. These are the settings entered in the *Parameters* dialog created via *getPropertyDescriptionList()*.

You can use an object's *on new* handler to set properties that may help track the information previously discussed. For example, you could set a property called *pParent* that contained the name of the parent script from which the object was being created. You can also use the *string()* function to convert the object reference into a string, then try to parse that for the parent script name and *referenceCount*: But the act of passing in the object reference and trying to parse it increases the *referenceCount*.

Avoid having an object refer to itself. An object with a property that contains a reference to the object itself will never be released from memory unless you specifically clear the property to "break the chain".

Child Object Properties

Although child objects themselves are not identical to property lists, the properties of a child object can be extracted using the property list functions, such as *count()*, *getAt()*, *getPropAt()*, and *setaProp()*. Refer to Example 12-10 and the *readProps()* utility in Example 6-19 of Chapter 6, *Lists*, that will extract an object's properties.

Example 12-12: Reading Properties of Child Objects

```
set gAdam = new (script "Eden", #male, "Adam", FALSE)
readProps(gAdam)
-- "This #instance has 3 properties"
-- "#pGender: male"
-- "#pName: Adam"
-- "#pKnowledge: 0"
```

To determine the value of any property associated with any Behavior attached to a sprite, you can use:

```
put the property of sprite whichSprite
```

For example, assuming sprite 3 has the "Eden" script attached from Example 12-10:

```
put the pName of sprite 3
-- "Adam"
```

If more than one attached Behavior contains the same property, the value will be returned for the first Behavior found with the specified property. If you want to

get a property of a specific Behavior, you can extract it from *the scriptInstanceList*, which contains a list of attached Behavior script instances.

```
put the pName of getAt (the scriptInstanceList of sprite 3, 2)
-- "Eve"
```

You can use global variables instead of properties to create values that are common across all instances of an object. Of course, the global will be universal throughout the entire movie, as would any global.

Behaviors versus Other Script Types

Now that you understand a bit about objects, let's revisit Behaviors. Ironically, Behaviors are most useful to users at opposite ends of the spectrum. Beginners can use prewritten Behaviors without understanding their inner workings, and experts can create their own powerful Behaviors. I'll assume you are somewhere in between and that most of the Behaviors in the Behavior Library are too simple for your needs, yet you have no idea how to construct your own Behavior or even when you should try.

What Is a Behavior Script?

When I first used Behaviors I thought of them as parent (that is, object-oriented) scripts tied to a sprite or frame. Now that I've used them for a while, I think of them as instantiated score scripts. Table 12-1 shows some important differences among these three types of scripts. For the purposes of this table, the term *score script* refers to simple Director 5 style score scripts (In D6, a score script is a Behavior is a score script.)

Table 12-1: Behaviors, Score Scripts, and Parent Scripts

Type	Instantiated?	Attached to	Script Type	User-configurable Properties?	Easy to Write?	Easy to Use?
Score Script	Yes[1]	Sprite or Frame	#score	No	Easiest	Medium
Behavior	Yes[1]	Sprite or Frame	#score	Usually	Hardest	Easiest
Parent Script	Manually[2]	Nothing	#parent	No	Medium	Hardest

[1] All score scripts are instantiated automatically by Director 6 when the sprite or frame to which they are attached is encountered, even though they might not have any properties.
[2] Parent scripts are manually instantiated by the programmer using *new()*.

Differences Among Behaviors, Sprite Scripts, and Frame Scripts

Behaviors are technically a type of score script. Like all score scripts, they are treated as either frame scripts or sprite script depending on whether they are attached to the script channel or a sprite channel. When designed to be attached

to a sprite, they predominantly respond to sprite-related events, such as *mouseUp*. When designed to be attached to a frame, they predominantly respond to frame-related events, such as *exitFrame*.

Differences Between Behaviors and Parent Scripts

Parent scripts and Behaviors are both object-oriented scripts. When a parent script or Behavior is used, Director creates an *instance* of it, which can be thought of as a copy, or clone, with its own set of values for each property (attribute).

Object-oriented means that the script acts as a template for a living, breathing entity with a life of its own. When a parent script is instantiated, a new child object is spawned. The child object owes it creation to the original parent script, yet it operates independently from the parent script and from any other siblings created from the same parent.

A Behavior is like a parent script that is used in the Score; it also has extra code to create a dialog that allows a user to customize properties easily. When a Behavior is encountered in the Score, Director creates a *script instance* that is analogous to a child object.

Behaviors are attached to either a sprite or a frame in the Score and created and disposed of automatically by Director when a sprite span begins or ends. They are initialized automatically by Director with the Parameters specified during authoring rather than via an *on new* handler.

Child objects of Parent scripts are created and disposed of manually by the programmer and aren't tied to sprites or frames. The association between a Behavior and the frame or sprite to which it is attached is *automatic*.

Use Behaviors to add some functionality to an entity in the Score. Suppose you want to alert the user whenever they are inactive for five minutes. You can either attach an appropriate timeout Behavior to the frame or to the sprite used as a prompt. When the timeout occurs, change the sprite prompt or play an audio warning.

Behaviors require the programmer to do *more* work, but the user to do *less* work. When creating a Behavior, you must specify all the information that Director needs to create a dialog box that prompts the user for the necessary properties.

Anatomy of a Behavior

Behaviors don't *require* anything beyond what is in any other sprite script or frame script. A sprite Behavior may perform any initialization in an *on beginSprite* handler and often traps mouse events using mouse handlers, such as *on mouseUp*. Frame Behaviors also often perform any initialization in an *on beginSprite* handler and do their remaining work in *on exitFrame* handlers. (Don't perform frame initialization in a *prepareFrame* handler because it is called every frame.)

Handlers in Behaviors automatically receive *me* (which contains a reference to the current script instance) as the first argument. Movie scripts and cast scripts are not instantiated, and their handlers don't receive *me*.

Behaviors often include *getPropertyDescriptionList* and *getBehaviorDescription* methods. The optional *runPropertyDialog* method is less common.

The *getBehaviorDescription*, *getPropertyDescriptionList,* and *runPropertyDialog* events are completely unlike standard Director events. They are *not* sent when the movie *is* playing, but they *are* sent when it is *not* playing. *RunPropertyDialog* and *getPropertyDescriptionList* can be called frequently when using the Behavior Inspector, and they can wreak havoc if they do not return a proper list. You should not set breakpoints in these handlers, as it is easy to create an infinite loop. They should call only built-in Lingo commands, and not custom handlers, because at compile time such handlers are not yet valid. You have been warned.

Table 12-2: Behavior-Related Event Handlers

Message	Description
on getBehaviorDescription	Called when the Behavior is highlighted in the Behavior Inspector. The return string appears in the bottom pane of the Behavior Inspector.
on getPropertyDescriptionList	Called when the script is compiled, attached to a sprite or frame, or the *Parameters* button is used in the Behavior Inspector. Also called to retrieve default parameters for use by *on runPropertyDialog*.
on runPropertyDialog	Call whenever the *Parameters* dialog would be displayed. The default property values returned by *on getPropertyDescriptionList* are sent to the *on runPropertyDialog* handler, which can modify them to return a custom list of properties.

The GetBehaviorDescription Method

The *on getBehaviorDescription* method should return a text string describing the Behavior. The Help message is displayed in the Description Pane of the Behavior Inspector. It typically describes the parameters the Behavior accepts and specifies whether it is intended as a sprite script or a frame script.

Example 12-13: The GetBehaviorDescription Method

```
on getBehaviorDescription me
  return "This is some help text"
end
```

The GetPropertyDescriptionList Method

Director uses the *on getPropertyDescriptionList* method to create a dialog box to prompt the user for property settings. The method should return a *property* list of only those properties that are *user-settable* for a given Behavior. For *each* property, specify the property's name, followed by attributes that control its appearance in the dialog box. Example 12-14 shows a sample property list with only one user-settable property. (The *#range* attribute is optional and has two possible formats.)

Example 12-14: The GetPropertyDescriptionList Method

```
on getPropertyDescriptionList me
  set propList = [¬
   #propertyName: ¬
   [#comment: "user prompt", ¬
    #format: dataType, ¬
   {#range: [#min:minValue, #max:maxValue] |¬
            [value1, value2, ...valuen],  ¬}
    #default: defaultValue] ¬
   ]
  return propList
end
```

#propertyName
: The name of the property variable that you want to let the user set. It should also be defined at the top of the Behavior script, using:

  ```
  property propertyName
  ```

#comment: "userPrompt"
: The text defined by *#comment* will appear in the dialog box presented to the user. Keep the text short and descriptive.

#format: dataType
: The *#format* entry tells Director what data type the user should be allowed to input. The allowed values are shown in Table 12-3.

#default: defaultValue
: The *#default* entry specifies an initial value for *#propertyName* and should be of the type specified by *#format*. The *#default* can be set to a variable's name that will be evaluated at runtime.

#range
: The optional *#range* entry can be *either* a linear list of enumerated values (which will appear as a pop-up menu) or a property list specifying a *#min* and *#max* range.

The *getPropertyDescriptionList* example in Director's online Help does *not* document the *#range* entry. It is also missing a comma after `#fieldNum` in the line "`addProp description, #fieldNum,[#default:1,`."

The title of the *Property* dialog is set to the cast member name (or the cast member number if the name is EMPTY). Use a descriptive cast member name to remind yourself what the Behavior does.

Because the Behavior parameters popup uses the MUI Xtra, it is beholden to the same limitations. You can fit only about 15 parameters on a 640-by-480 screen before the dialog fails to appear. See *http://www.updatestage.com/previous/970801.html#item3* for details. A Behavior can have many properties, but the *Parameters* dialog may fail if more than 15 of those properties are specified in the list returned by *on getPropertyDescriptionList*. Implement a custom MUI dialog via the *on runPropertyDialog* handler if necessary (see the downloadable Chapter 21, *Custom MUI Dialogs*).

The #format Code

The *#format* of a property in the property description list determines how the user is prompted and what type of data he or she is allowed to enter. Table 12-3 and Table 12-4 show the *#format* codes that let you select from a pop-up list of items of the specified type, such as bitmap cast members, sound cast members, or marker labels. For example, if you are writing a Behavior that requires a sound, you might let the user pick that sound from a list of sound cast members by using a *#format* of *#sound* for the property of interest.

Using a *#format* such as *#graphic* that may encompass hundreds of cast members will create a popup with hundreds of entries, taking considerable time and possibly crashing the system.

Table 12-3: #format Codes for Cast Member Types

Message	**Sent When**
#format	Matching *the type of member* property
#bitmap	#bitmap cast members only
#button	#button cast members only
#digitalVideo	#digitalVideo cast members only (excludes #quickTimeMedia)
#field	#field cast members only (not #richText)
#filmLoop	#filmLoop cast members only
#graphic	#bitmap, #btned, #button, #digitalVideo, #field, #filmLoop, #movie, #ole, #picture, #PopMenu, #richText, #shape, #SWA (any cast member type that can be used in a sprite channel)
#member	all cast member types (those listed above for #graphic, plus #palette, #script, #sound, and #transition)
#movie	#movie cast members only (not movie scripts)

Table 12-3: #format Codes for Cast Member Types (continued)

Message	Sent When
#ole	#ole cast members only
#palette	built-in palettes, plus #palette cast members
#picture	#picture cast members only
#richText	#richText cast members only
#shape	#shape cast members only
#script	#script cast members only
#sound	#sound cast members only (not #SWA)
#transition	built-in transitions, plus #transition cast members

Note that *getPropertyDescriptionList* doesn't recognize *#ActiveX*, *#btned*, *#flash*, *#SWA*, *#quickTimeMedia*, *#text*, or *#xtra* as separate cast member types.

Table 12-4 shows the *#format* options that don't pertain to a cast member types, but rather to "pure" data types (floats, integers, Booleans, symbols, and strings) and other Director entities (cursors, markers, and inks).

Table 12-4: Non-Castmember Behavior #format Codes

#format	User sees	#default	#range
#boolean	Checkbox	`TRUE` or `FALSE`	N/A
#cursor	Pop-up menu	Installed cursors[1]	None or enumerated
#float	Entry field, slider, or pop-up menu[2]	0.0 or your choice	#min/#max, none, or linear list
#ink	Pop-up menu	Name of your choice	None or enumerated
#integer	Entry field, slider, or pop-up menu[2]	0 or your choice	#min/#max, none, or linear list
#marker	Pop-up menu	`previous`, `loop`, and `next`	`previous`, `loop`, `next`, plus any custom marker labels
#string	Entry field or pop-up menu[3]	`EMPTY` or your choice	None or linear list
#symbol	Entry field or pop-up menu[3]	`EMPTY` or your choice	None or linear list

[1] List of available cursor resources varies between Mac and Windows, but may include Arrow, I-Beam. Crosshair, Crossbar, Watch, Blank, Help, Finger, Hand, Closed Hand, No Drop Hand, Copy Closed Hand, Pencil, Eraser, Select, Bucket, Lasso, Dropper, Air Brush, Zoom In, Zoom Out, Vertical Size, Horizontal Size, and Diagonal Size. List does not include custom 1-bit cast members used as cursors (use #bitmap instead).
[2] If a #min/#max #range is specified the user sees a slider. If a linear list is used for #range, the user sees a popup menu. If no range is specified, the user sees a text entry field.
[3] User sees entry field or popup menu depending on #range as per footnote 2.

Here is an example *getPropertyDescriptionList()* if you were to turn the *Eden* script from Example 12-10 into a Behavior.

Example 12-15: Behaving Yourself in Paradise

```
property  pGender, pName, pKnowledge

on eatApple me
  set pKnowledge = TRUE
end

on getPropertyDescriptionList me
  set propList = [[LC]
   #pGender:[#default: #male,  #format: #symbol, ¬
              #comment: "Gender", #range:[#male, #female]], ¬
   #pName:   [#default: EMPTY,  #format: #string,¬
              #comment: "Person's Name"], ¬
   #pKnowledge: [#comment: "Tree of Knowledge",  ¬
              #default: FALSE, #format: #boolean] ¬
  ]
  return propList
end
```

The *on getPropertyDescriptionList* is called only at authoring time to store the default properties for a Behavior. Those properties are applied when the script is instantiated at runtime. Don't forget to *return* the property list you've built.

The RunPropertyDialog Method

Despite its name, the *on runPropertyDialog* handler is never called at runtime. It is called only when a Behavior's *Parameter* dialog would otherwise appear during authoring. If present in a Behavior, the *on runPropertyDialog* handler is called to set values for the properties without prompting the user via the *Parameter* dialog. It receives a list of default values of the properties returned from *on getPropertyDescriptionList*. In Example 12-16, `defaultProps` is *[#pGender: #male, #pName: "", #pKnowledge: 0]* before the handler is called, and *[#pGender: #male, #pName: "Seth", #pKnowledge: 0]* afterwards.

Example 12-16: The RunPropertyDialog Method

```
on runPropertyDialog me, defaultProps
  set the pName of defaultProps = "Seth"
  return defaultProps
end
```

The Lingo Dictionary and online Help have incorrect entries for *runPropertyDialog*.

Behavior and Parent Script Lingo

Table 12-5 covers Lingo pertaining to Behaviors and parent scripts.

The text of scripts is limited to 32,000 characters, so it easy to run out of room when creating complex objects.

You can use the ancestor property to create objects that use multiple scripts, but the management can get annoying. Hopefully, Macromedia will address the 32,000 characters *scriptText of member* limit in Director 7.

Table 12-5: Behaviors and Parent Scripts Lingo

Command	**Usage**
the actorList	A list of object instances that receive the *stepFrame* message. Use to send messages to child objects each time playback head moves.

ancestor	A property of a child object that points, not to the parent script, but to a "grandparent" script of your choosing.
	`property ancestor`
	`set ancestor = new (script "Ancestor Script")`

birth (script "*Parent Script*")	Obsolete. Use *new()* instead.
call (#*handlerName*, *script* | *scriptInstance* | *objectList* {, *args*})	Sends a custom message to a one or more scripts, script instances, or child objects.
callAncestor (#*handlerName*, *script* | *scriptInstance* | *objectList* {, *args*})	Sends a custom message to the ancestor of one or more script instances or child objects.
the currentSpriteNum	Indicates the current sprite number from within a Behavior attached to a sprite.
getBehaviorInitializers(*spriteNum*, *behaviorNum*)	Returns the list of default property values passed to *on runPropertyDialog*. Requires *UI Helper* Xtra, included with D6.5.
getBehaviorMemRef(*spriteNum*, *behaviorNum*)	Returns an absolute cast member reference of the specified behavior attached to *spriteNum* (or 0). Requires *UI Helper* Xtra, included with D6.5.
me	Identifies the current script instance. Returned by *new()* when the child object is created and used as a parameter to Behavior and object methods.

Table 12-5: Behaviors and Parent Scripts Lingo (continued)

Command	Usage
new()	Used to create a new instance of a parent script or Behavior. `set childObj = new (script "ParentScript" {, args...})`
on getPropertyDescriptionList	Returns a property list defining all the user-settable properties of a Behavior. See Table 12-2.
on getBehaviorDescription	Returns a text string describing the Behavior, displayed in Behavior Inspector as Help text. See Table 12-2.
on new	Handler called when a parent script is instantiated. Must return a script instance (*me*)
on runPropertyDialog	Changes Behavior's properties without user intervention. Suppresses *Parameter* dialog. See Table 12-2.
on stepFrame	Handler to perform some action each time the *stepFrame* message is sent to *the actorList*.
property *propertyVar*	Defines a semi-private variable for a Behavior or parent script.
script "*Parent*" or script *scriptNum*	Used with *new()* to refer to a Parent Script.
the scriptInstanceList of sprite	A list of Behavior instances attached to a given sprite. Available only while Director is running. See *getBehaviorMemRef()*.
the scriptType of member	Script type should be set to #score for Behaviors and #parent for parent scripts.
send (object, #*message*)	Unsupported variant of *call()*.
sendAllSprites (#*message* {, *args*})	Sends a message to all sprites in the current frame.
sendAncestor (*object*, #*message*)	Unsupported variant of *callAncestor()*.
sendSprite (*whichSprite*, #*message* {, *args*})	Sends a message to a particular sprite in the current frame.
the spriteNum of me	Indicates the current sprite number from within a Behavior attached to a sprite.

Handler Evaluation with Behaviors

New Sprite Events Sent to Behaviors Used as Sprite Scripts

See Chapter 2 for a full description of events. In Director 6, Score scripts attached to the sprite channels now receive *new*, *enterFrame*, and *exitFrame* messages (plus many other new messages). These handlers would not have been called when playing in Director 5. The Director 6 CD includes a cleaner utility in the `Goodies\Movies\Cleaner` folder, and D6.0.2 now warns about these when updating from Director 5 movies. (See the Director 6 *ReadMe* file.)

For backwards compatibility the Shockwave for D6 plug-in does not send *enterFrame* and *exitFrame* messages to sprites if the movie file being played is of pre-Director 6 vintage. It does send the *new* message to sprites however.

Handler Execution, The ScriptInstanceList, and the SpriteNum of Me

As described in Example 2-11 in Chapter 2, the *new* event is sent to sprites before the *beginSprite* event. If a Behavior defines an *on new* method, it is called before *the scriptInstanceList* is populated with any Behavior instances, and before the property values are assigned. Therefore, you can't manipulate *the scriptInstanceList* or assume that Behavior properties have been defined in an *on new* handler.

If a score script is going to be used as both a sprite Behavior *and* instanced explicitly via the *new()* function, then the *on new* handler should declare and set the *spriteNum of me* property. (It is set automatically when an instance of the script is created for a Behavior, but not if *new()* is called explicitly.

Refer to the Director 6 *ReadMe* and Director 6.0.1 *Updates* file for more details.

The ActorList

The *stepFrame* message is sent only to items in the *actorList*. In Director 6, sprites receive *prepareFrame*, *exitFrame*, and *enterFrame* events, but *the actorList* can still be used to notify Parent scripts and child objects (which don't receive events by default) when the playback head moves. Refer to "The ActorList" in Chapter 2 for more details. Note that each item on *the actorList* can have its own *on stepFrame* handler to take appropriate action when the playback head advances.

The following can be used to add and remove items from *the actorList*.

Example 12-17: The ActorList

```
on addToActorList dummy, object
  -- only add objects to the actorList
  if not objectP(object) then
    alert "Not an object" && object
    exit
  end if
  -- Don't add it if it is already on the list
  if getPos (the actorList, object) = 0 then
    add the actorList, object
  end if
end addToActorList

on removeFromActorList dummy, object
  -- Remove the item from the actorList
  set offset = getPos (the actorList, object)
  if offset then
    deleteAt (the actorList, offset)
  end if
end removeFromActorList
```

See Example 1-34 under "Special Treatment of the First Argument Passed" in Chapter 1 for details on why the above uses a dummy parameter instead of passing `object` as the first parameter.

The following uses the utilities in Example 12-17 cause a Behavior instance to automatically add its sprite to (and delete its sprite from) *the actorList*. The sprite will then receive *stepFrame* events.

```
on beginSprite me
  addToActorList (void, the spriteNum of me)
end

on endSprite me
  removeFromActorList (void, the spriteNum of me)
end
```

To clear the actorList entirely, you can use *deleteAll(the actorList)* or simply set it to `[]`.

The *clearGlobals* command also clears *the actorList* in D6, although it didn't do so in D4 or D5.

Adding Behaviors at Runtime

The scriptInstanceList of sprite property is a Lingo list of instantiated Behaviors. Until the playback head enters the sprite of interest, *the scriptInstanceList* is the EMPTY list ([]). See "Script Instances" in Chapter 2 for details.

You can't set *the scriptInstanceList* during authoring or even a Score Recording session because it is a list of instances, not script numbers. *The scriptNum of sprite* returns only the first attached Behavior, but you can use the *getBehaviorMemRef()* function included with the *UI Helper* Xtra (included with D6.5) to count the number of attached Behaviors and determine their member numbers.

Example 12-18: Getting Attached Behaviors

```
on getBehaviors spriteNum
  set n = 1
  set memList = [ ]
  repeat while (TRUE)
    -- This requires the UI Helper Xtra
    if getBehaviorMemRef (spriteNum, n) = 0 then
      exit repeat
    else
      add memList, member getBehaviorMemRef (spriteNum, n)
      set n = n + 1
    end if
```

Example 12-18: Getting Attached Behaviors (continued)

```
  end repeat
  return memList
end getBehaviors
```

You can add Behaviors to a sprite at runtime by adding a script instances to its *scriptInstanceList* (but only if at least one Behavior was already attached).

Example 12-19: Adding Behaviors At Runtime

```
set newBehavior = new(script"BehaviorScript" {, args})
add (the scriptInstanceList of sprite n, newBehavior)
```


Hopefully this chapter has dispelled the mysterious aura surrounding object-oriented programming and Behaviors. You should have recognized the deep parallels between programmer-defined objects and Director's built-in entities (members, sprites, windows, lists, and so on). All such entities have properties that can be manipulated without intimate knowledge of their internal structure.

Although some zealots create objects for *everything*, then create more objects to manage their objects, I use OOP selectively. Develop your own style. Regardless of your enthusiasm for OOP, you should add it to your arsenal. If you've understood this chapter, you'll also understand that some tasks cry out for OOP. Properly applied, it will make your code easier to write and more maintainable.