

Squeak

Learn Programming with Robots

STÉPHANE DUCASSE

Squeak: Learn Programming with Robots

Copyright © 2005 by Stéphane Ducasse

Lead Editor: Jonathan Hassell

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis,
Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Assistant Publisher: Grace Wong

Project Manager: Kylie Johnston

Copy Manager: Nicole LeClerc

Copy Editor: David Kramer

Production Manager: Kari Brooks-Copony

Production Editor: Kelly Winkist

Compositor: Diana Van Winkle, Van Winkle Design Group

Proofreader: Elizabeth Berry

Indexer: Valerie Perry

Artist: Diana Van Winkle, Van Winkle Design Group

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Library of Congress Cataloging-in-Publication Data

Ducasse, Stéphane.

Squeak : learn programming with robots / Stéphane Ducasse.

p. cm.

ISBN 1-59059-491-6

1. Robots--Programming. I. Title.

TJ211.45.D83 2005

629.8'925117--dc22

2005013248

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

Contents at a Glance

Foreword.....	xvii
About the Author.....	xxiii
Acknowledgments.....	xxv
Preface.....	xxvii

PART 1 ■ ■ ■ Getting Started

■ CHAPTER 1	Installation and Creating a Robot.....	3
■ CHAPTER 2	A First Script and Its Implications.....	13
■ CHAPTER 3	Of Robots and Men.....	29
■ CHAPTER 4	Directions and Angles.....	37
■ CHAPTER 5	Pica’s Environment.....	51
■ CHAPTER 6	Fun with Robots.....	61

PART 2 ■ ■ ■ Elementary Programming Concepts

■ CHAPTER 7	Looping.....	77
■ CHAPTER 8	Variables.....	87
■ CHAPTER 9	Digging Deeper into Variables.....	101
■ CHAPTER 10	Loops and Variables.....	109
■ CHAPTER 11	Composing Messages.....	119

PART 3 ■ ■ ■ Bringing Abstraction into Play

■ CHAPTER 12	Methods: Named Message Sequences.....	135
■ CHAPTER 13	Combining Methods.....	149
■ CHAPTER 14	Parameters and Arguments.....	155
■ CHAPTER 15	Errors and Debugging.....	167
■ CHAPTER 16	Decomposing to Recompose.....	183
■ CHAPTER 17	Strings, and Tools for Understanding Programs.....	197

PART 4 ■ ■ ■ Conditionals

■ CHAPTER 18	Conditions.....	209
■ CHAPTER 19	Conditional Loops	221
■ CHAPTER 20	Boolean and Boolean Expressions.....	233
■ CHAPTER 21	Coordinates, Points, and Absolute Moves.....	243
■ CHAPTER 22	Advanced Robot Behavior.....	261
■ CHAPTER 23	Simulating Animal Behavior.....	269

PART 5 ■ ■ ■ Other Squeak Worlds

■ CHAPTER 24	A Tour of eToy	289
■ CHAPTER 25	A Tour of Alice	315
■ INDEX	337

Contents

Foreword.....	xvii
About the Author.....	xxiii
Acknowledgments	xxv
Preface	xxvii

PART 1 ■ ■ ■ Getting Started

■ CHAPTER 1	Installation and Creating a Robot	3
	Installing the Environment	4
	Installation on a Macintosh	4
	Installation under Windows	4
	Opening the Environment.....	5
	Tips for Installation	5
	First Interactions with a Robot	6
	Sending Messages to a Robot.....	7
	Creating a New Robot.....	9
	Quitting and Saving.....	10
	Installation Troubleshooting.....	10
	Summary	12
■ CHAPTER 2	A First Script and Its Implications	13
	Using a Cascade to Send Multiple Messages.....	14
	A First Script.....	15
	Squeak and Smalltalk.....	16
	Programming Languages.....	16
	Smalltalk and Squeak.....	17
	Programs, Expressions, and Messages.....	18
	Typing and Executing Programs	18
	The Anatomy of a Script.....	18
	About Pixels	19
	Expressions, Messages, and Methods	19
	Message Separation	21

Method	22
Cascade	22
Creating New Robots	22
Errors in Programs	23
Misspelling a Message Selector	24
Misspelling a Variable Name	24
Unused Variables	25
Uppercase or Lowercase?	25
Forgetting a Period	26
Words That Change Color	27
Summary	28
CHAPTER 3 Of Robots and Men	29
Creating Robots	30
Drawing Line Segments	31
Changing Directions	31
The ABC of Drawing	34
Controlling Robot Visibility	35
Summary	35
CHAPTER 4 Directions and Angles	37
Right or Left?	38
A Directional Convention	39
Absolute Versus Relative Orientation	40
The Right Angle of Things	42
A Robot Clock	45
Simple Drawings	46
Regular Polygons	47
Summary	48
CHAPTER 5 Pica's Environment	51
The Main Menu	51
Obtaining a Bot Workspace	52
Interacting with Squeak	53
Using the Bot Workspace to Save a Script	54
Loading a Script	55
Capturing a Drawing	55
Message Result	57

Executing a Script	58
Hints	58
Two Examples	58
Summary	60
CHAPTER 6 Fun with Robots	61
Robot Handles	62
Pen Size and Color	62
More about Colors	63
Changing a Robot's Shape and Size	64
Drawing Your Own Robot	66
Saving and Restoring Graphics	67
The "Save Graphics" Handle	68
Retooling the Robot Factory	68
Graphics Operations Using Scripts	69
Summary	73
PART 2 ■ ■ ■ Elementary Programming Concepts	
CHAPTER 7 Looping	77
A Star Is Born	78
Loops to the Rescue	79
Loops at Work	80
Code Indentation	80
Drawing Regular Geometric Figures	81
Rediscovering the Pyramids	82
Further Experiments with Loops	84
Summary	86
CHAPTER 8 Variables	87
Brought to You by the Letter A	88
Variations on the Theme of A	88
Variables to the Rescue	90
Declaring a Variable	90
Assigning a Value to a Variable	90
Referring to Variables	91
And What About Pica?	91

Using Variables	91
The Power of Variables	92
Expressing Relationships Between Variables	93
Experimenting with Variables	94
The Pyramids Rediscovered	95
Automated Polygons Using Variables	96
Regular Polygons with Fixed Sizes	98
Summary	98
CHAPTER 9 Digging Deeper into Variables	101
Naming Variables	102
Variables as Boxes	102
Assignment: The Right and Left Parts of :=	103
Analyzing Some Simple Scripts	104
Summary	108
CHAPTER 10 Loops and Variables	109
A Bizarre Staircase	110
Practice with Loops and Variables: Mazes, Spirals, and More	113
Some Important Points for Using Variables and Loops	115
Variable Initialization	116
Using and Changing the Value of a Variable	116
Advanced Experiments	117
Summary	118
CHAPTER 11 Composing Messages	119
The Three Types of Messages	120
Identifying Messages	120
The Three Types of Messages in Detail	122
Unary Messages	122
Binary Messages	123
Keyword-Based Messages	123
Order of Execution	124
Rule 1: Unary > Binary > Keywords	125
Rule 2: Parentheses First	127
Rule 3: From Left to Right	129
Summary	131

PART 3 ■ ■ ■ Bringing Abstraction into Play

■ CHAPTER 12	Methods: Named Message Sequences	135
	Scripts versus Methods	136
	How Do We Define a Method?	137
	A Class Bot Browser	138
	Creating a New Method Category	139
	Defining Your First Method	140
	What's in a Method?	142
	Scripts versus Methods: An Analysis	143
	Returning a Value	144
	Drawing Patterns	145
	Summary	147
	Glossary	147
■ CHAPTER 13	Combining Methods	149
	Nothing Really New: The Square Method Revisited	150
	Other Graphical Patterns	150
	What Do These Experiments Tell You?	151
	Squares Everywhere	153
	Summary	154
■ CHAPTER 14	Parameters and Arguments	155
	What Is a Parameter?	156
	A Method for Drawing Squares	156
	Practice with Parameters	158
	Variables in Methods	159
	Experimenting with Multiple Arguments	160
	Parameters and Variables	162
	Arguments and Parameters	164
	About Method Execution	165
	Summary	166
■ CHAPTER 15	Errors and Debugging	167
	The Default Value of a Variable	168
	Looking at Message Execution	169
	A First Look at the Debugger	172

Stepping through the Stack	175
Fixing Errors	178
Example 1	179
Example 2	179
Summary	181
CHAPTER 16 Decomposing to Recompose	183
Mazes and Spirals	184
Centered Squares	184
Spirals	186
Golden Rectangles	189
A One-Line-per-Rectangle Solution	190
Tiling	194
Summary	195
CHAPTER 17 Strings, and Tools for Understanding Programs	197
Strings	198
Communicating with the User	198
Strings and Characters	199
Strings and Numbers	201
Using the Transcript	202
Generating and Understanding a Trace	203
Summary	206
PART 4 ■ ■ ■ Conditionals	
CHAPTER 18 Conditions	209
A Robot's True Colors	210
Adding a Trace to See What Is Going On	211
The Value Returned by a Method	212
Conditional Expressions with Only One Branch	213
Choose the Right Conditional Method	214
Nesting Conditional Expressions	214
Robot Coloring with Three Colors	214
Learning from Your Mistakes	216
Interpreting a Tiny Language	218
Further Experiments	219
Summary	220

CHAPTER 19	Conditional Loops	221
	Conditional Loops	222
	An Example	222
	Experiences with Traces	224
	Stopping an Infinite Loop	226
	Deeper into Conditional Loops	228
	A Simple Interactive Application	228
	When to Use Square Brackets	230
	Summary	230
CHAPTER 20	Boolean and Boolean Expressions	233
	Boolean Values and Boolean Expressions	234
	Boolean Values	234
	Boolean Expressions	234
	Combining Basic Boolean Expressions	235
	Negation (not)	236
	Conjunction (and)	236
	Alternation (or)	237
	All of the Above	237
	Some Smalltalk Points	237
	Missing Parentheses (a Frequent Mistake)	238
	A Case Study	239
	Using the Debugger	240
	Understanding the Problem	240
	Similar Problems and Solutions	241
	Summary	241
CHAPTER 21	Coordinates, Points, and Absolute Moves	243
	Points	244
	Using Grids	246
	A Source of Errors with Points	247
	Decomposing $50@60 + 200@400$	248
	Decomposing $(50@60) + (200@400)$	248
	Absolute Moves	248
	Relative versus Absolute Motion	249
	Some Experiments	251
	Translations	252
	Translating Triangles	253
	Flying Geese	254

Absolute Moves at Work	255
Loops and Translations	257
Further Experiments	259
Summary	259
CHAPTER 22 Advanced Robot Behavior	261
Obtaining a Robot's Direction	262
Pointing in a Direction	262
Distance from a Point	263
Back in the Center of the Screen	264
Location If It Moved	264
In a Box	264
Heading toward a Point	265
Center versus Position	267
Summary	267
CHAPTER 23 Simulating Animal Behavior	269
Wandering	270
Separating Influences	271
Studying the Influence of the Length	272
Studying the Influence of the Side to Which the Animal Turns	273
Trapped in a Box	274
Following Borders	275
Flying to the Opposite Border	276
Random Direction	276
Introducing an Exit in the Box	277
Staying in a Healthy Environment	278
Further Experiments	280
Finding Food	280
Comparing Distance	280
Taking One's Bearings	283
Simulating Vision	284
Summary	286

PART 5 ■ ■ ■ Other Squeak Worlds

■ CHAPTER 24	A Tour of eToy	289
	Steering an Airplane.....	290
	Step 1: Drawing an Airplane	290
	Step 2: Playing with the Halo.....	291
	Step 3: Dragging and Dropping a Method to Create New Scripts ..	295
	Step 4: Adding Methods	296
	Joysticks in Action.....	297
	Step 1: Creating a Joystick	297
	Step 2: Experimenting with a Joystick.....	298
	Step 3: Linking the Joystick and the Script.....	298
	Creating an Animation	299
	Step 1: Creating the Holder	299
	Step 2: Drawing Animation Elements.....	300
	Step 3: Dropping the Pictures into the Holder.....	301
	Step 4: Creating a Simple Sketch Recipient of the Animation . . .	301
	Step 5: Creating a Script with lookLike	302
	Step 6: Displaying the Selected Animation Element	302
	Step 7: Changing the Currently Selected Element of a Holder . . .	303
	Another Way	304
	Cars and Drivers	305
	Step 1: Draw a Car and a Steering Wheel	305
	Step 2: Turning the Car in a Circle.....	305
	Step 3: Using the Wheel's Heading.....	306
	Step 1: Sensors	308
	Step 2: The Road	308
	Step 3: Conditions and Tests in eToy	309
	Step 4: Customizing Color-Based Tests	310
	Step 5: Adding Actions	311
	Some Tricks	311
	Running Several Scripts	312
	Clearing	312
	Creating a Tile.....	312
	Internationalization	314
	Summary	314

CHAPTER 25 A Tour of Alice	315
Getting Started with Alice	316
Interacting Directly with Actors	318
The Environment	320
Scripts	321
Analyzing a First Script	322
Moving, Turning, and Rolling	323
Actor Parts	324
Other Operations	325
Getting Bigger	325
Quantified Moves	326
Standing Up	326
Coloring	326
Destruction	326
Visibility	326
Absolute Moves and Rotations	326
Pointing At	327
Relative Placement of Actors	327
Time-Related Actions	327
Animation	328
Your Own Wonderland	329
Multiple Cameras and Other Special Effects	330
Alarms	332
Introducing User Interaction	332
Hidden Aspects of Alice and Pooh	333
Mapping 2D Morphs to 3D	333
Pooh: Generating 3D Forms from 2D	335
Summary	336
INDEX	337

Foreword

By Alan Kay
President, Viewpoints Research Institute, Inc. & Sr. Fellow,
The Hewlett-Packard Company

The Future of Programming As Seen from the 1960s

I started graduate school (at the University of Utah ARPA Project) in November 1966, and it is interesting to look back on the world of programming as I surveyed it at that time.

The amazing Jean Sammit (who was an inventor of programming languages and their first historian, as well as being the first woman president of the ACM) was able to count about 3,000 programming languages that were extant by the late 1960s. Much was going on, and some of it was of great import and interest.

Algol 60, as Tony Hoare pointed out, “was a great improvement, especially on its successors!” It had many surface virtues, including a stronger feeling for contexts and environments for meaning in a programming language, and one remarkable feature for its day—call by name—which allowed its programmers a range of expression very similar to the language designers themselves. For example, one could write procedures that would have the same meanings and actions as the control statements in the language:

```
for (i, 1, 10, print(a[i]))
```

where the first and fourth parameters would be marked `name` and thus bundled into an expression that correctly remembered the hierarchical namespace context of its variables, but could be manipulated and executed from inside the body of the `for` procedure. Not even the original LISP did this correctly at first!

And there was a little-known syntactic variant in the Algol 60 official syntax that encouraged a more readable form for made-up procedures. This allowed a comma in a procedure call to be replaced by the following construct:

```
) : <some comment> (
```

and this would allow the preceding call to be written as follows:

```
for (i): from (1): to (10): do ( print( a[i] ) )
```

Do this with a nice display or IBM Executive typewriter made into a terminal (as JOSS had), and you would get

```
for (i): from (1): to (10): do ( print( a[i] ) )
```

which looks a lot like the Algol base language but done as a meta-extension by the programmer for the benefit of other programmers.

Perhaps the single most profound set of language ideas and representations happened earlier than Algol, but took much longer for most computer people to understand (and many never did), in part because of the different and difficult-to-read notation (for outsiders at least), and because many of LISP's greatest contributions were "really meta." One of the great contributions of LISP was its evaluator written in itself in a half page of code. This was a kind of "Maxwell's Equations" for programming, and it allowed many things to be thought about that were essentially unthinkable in more normal approaches.

LISP itself was driven into existence to be the programming system for an interactive commonsense agent—The Advice Taker—that could take the wishes of a human user given in normal vernacular and turn them into computer processes that would carry out those wishes. Some very interesting intermediate languages, such as FLIP, and attempts at doing some of the Advice Taker properties, such as PILOT, were created in the mid 1960s.

Sketchpad was perhaps the most radical of the early systems because it tried to leap all the way to a reasonable interactive framework for people who wanted to use the computer for what it was best suited: interactive simulations of all kinds. The three cosmic contributions of Sketchpad were

- The first usable approach to interactive computer graphics
- A real object structure for all of its entities
- A nonprocedural way to program in terms of the desired end results, where the system could employ various automatic problem-solving processes to come up with the desired results

This was helped tremendously by a "tolerance approach" to solving constraints, which instead of trying for perfect logic/symbolic solutions of the sets of constraints instead tried to solve the constraints within global tolerances. This approach allowed many important problems to be dealt with that are still difficult or intractable symbolically today.

JOSS was a very different cup of tea: it did "almost nothing" (basically numerical calculations using numbers and array structures), but what it did do was done perfectly and in the form of what is still one of the great user interface designs in history.

A Programming Language was the name of a book by Kenneth Iverson that took a highly mathematical approach to programming via functions and metafunctions expressed as a kind of algebra. In those days the language was called "Iverson." An actual system in which you could program a computer was still just an IBM rumor at the time, but many paper programs were written using these ideas. The best thing about Iverson was that it really paid off if you thought of it as mathematical transforms and relationships, and didn't worry about how many operations would be required. Not worrying about number of operations was almost unthinkable in those days of 1 MHz clocks on multimillion-dollar building-sized computers, so Iverson and LISP were both very liberating vehicles for thinking ahead to the future, when machines would be smaller physically, and larger and faster logically.

The Simula designers wanted to model large, complex dynamic structures and realized that Algol blocks would do the job if you could cut them loose from Algol's hierarchical control structure. In the creation of Simula I in the mid 1960s, they were able to see that their ideas had great relevance to the language and its programming, and when they did Simula 67 they could replace many formerly built-in data types, such as `string`, with a Simula 67 class.

The idea of extending the syntax, semantics, and pragmatics of programming languages constituted an entire genre of investigation in the mid-to-late 1960s. One of the reasons for this is that it had become abundantly clear that programming was going to be difficult to scale, and that scalability in most dimensions was going to be critical to the health of computing. Where complexity is a central issue, architecture dominates materials. This realization started to make programming appear as something different from math, and it started to reveal itself as a new form of engineering. There were calls for the formation of a discipline to be called “software engineering” and to have a conference to try to figure out what this might mean (how to cope if you can’t just do math?).

ARPA Information Processing Techniques Office (IPTO) was in full swing by the time I went to graduate school in 1966, and it had already made some great starts toward its collective dream of having interactive computing for everyone pervasively connected via an “intergalactic network.” Just how to create this network (which had huge scaling requirements) generated some of the best systems thinking of the time, and was an important part of my own thinking about the future of programming.

The ARPA funders were wise and did not turn the vision of their dream into funding goals, but instead tried to find and fund talents that had their own ideas about what the dream meant and how it could be done. This resulted in about 17 sites in universities and companies, most of which had come up with very interesting and different designs and demos. This constituted a community of both “agreement and argument” that made everyone in it much smarter than they were before they joined the great dream.

Of course, given Jean Sammit’s 3,000 languages, there is much I haven’t mentioned, and much interesting design that happened from 1967 to the end of the decade that has to be omitted here. To pick just five developments of particular relevance to the readers of this book, I would choose the conception of objects that I came up with, and how they were supposed to be useful to end users of personal computers; Carl Hewitt’s PLANNER system, which was the most cohesive system for doing “programming as reasoning”; Ned Irons’s IMP system, which represents perhaps the first really useful completely extensible language; and Dave Fisher’s Control Definition Language, which illuminated extensibility in general and with respect to control structures in particular.

My background was in mathematics and molecular biology (I worked my way through school as a journeyman programmer) and in the arts. Circumstances forced me to try to understand Sketchpad, Simula, and the proposed ARPA intergalactic network in my first week in graduate school, and the reaction I had was cataclysmic. They were similar in some ways and very different in others, but they were different species of the same genus if one took both a biological and mathematical perspective. Biologically, they were “almost cells” crying out to be cells. Mathematically, they were “almost algebras” crying out to be algebras. So my initial fusion of these metaphors with computing was that you could make everything from entities that were logical computers that could send messages (which would also have to be logical computers). The logical computers would act the part of cells, and the protocols devised could be very algebraic—what today is (incorrectly) called *polymorphism*. This would result in great simplicity and scalability at the “materials level,” and would open the door for advancements in simplicity and scalability at the “expression level” where the programmer lived.

Several years later I found Hewitt’s PLANNER, and realized that it was the basis of a way to get programs to be both more meaningful and more scalable. (Many of the ideas of PLANNER also turned up in the later language called Prolog.) It was pretty clear that trying to send messages that were goal-oriented could greatly help scalability, in part because there are far

more ways to try to satisfy goals than goals (think of sorting as a goal versus all the ways to sort), and this separation could have great benefits in keeping programs more meaningful and less about optimizations mixed in with the meanings.

Meanwhile, the extensible language IMP had appeared, and there were several clever ideas that allowed it to be practical and not just wallow in its own meta-ness.

And, in parallel to the thesis I was working on about personal computers and object-oriented systems for all levels of users, Dave Fisher was working on a very nice complementary set of ideas about how to make control structures extensible via being able to add new meanings dynamically to a LISP-style meta-interpreter.

LOGO, the first great programming language for children, was a happy combination of JOSS and LISP, by Papert, Feurzig, Bobrow, and others at BBN. This opened up the idea of children as very important end users of the powerful ideas of computing, and changed my idea of computing from a tool or vehicle to a *medium* of expression that had a similar cosmic destiny to that of the printing press.

These five systems and the invitation to help start up Xerox PARC were the impetus for Smalltalk, and are most noticeable in the first versions of Smalltalk.

Looking back from today, it is striking that

- The level of expression in today's programming is so low (really back around 1965 for most of it), and very few programmers today program even at the level of what was possible in LISP and/or Smalltalk in the 1970s.
- Smalltalk has not changed appreciably since it was released as Smalltalk-80 in the early 1980s, even though it contains its own metasystem and is thus very easy to improve.
- Moore's Law from 1965 turned out to be pretty much correct, and we can now build huge HW and SW systems, yet they are very fragile because the scalable concepts beyond simple objectness have not been added (i.e., we perhaps have cells, but no concept of even tissues, or how to build/grow multicellular organisms).
- The Internet turned out to be a very successful expression of a radical approach to architecture and scaling, yet no software/programming system is set up to allow programmers to express Internet-like systems (what would the programs for the exemplary systems of Google and Amazon look like in such a new kind of programming system?).

What happened to progress in the last 25 years? And why is Squeak essentially just a free Smalltalk, if we desperately need progress?

In 1995 the Internet had gotten mature enough for us to try some experiments with media that we'd long wanted to do. And the Java (and other programming systems) of the time (and today) missed pretty badly in being flexible, meta, and portable enough to serve as a vehicle. Since we had done Smalltalk once before, and had written a book about how to do a complete such system, it made some sense to take a year to make a free, controllable Smalltalk and release it on the Internet (in fact, it took about nine months). The idea was that Squeak should not even be the vehicle so much as the factory for a much better twenty-first-century language.

However, programming systems in which programmers can program often take on a life of their own, and much of the Squeak open source movement and interest is in precisely a free Smalltalk with a media system that is highly portable. I think it is safe to say that most of the Squeak community is dedicated to making this Smalltalk more useful and accessible, and not

devoted to making something so much better as to render Smalltalk obsolete (a fate I would dearly love to see happen).

So, I would like to encourage the readers of this excellent new book to not think of Smalltalk as a bunch of features from the vendor gods that must be adhered to, but as a system that is capable of great extension in all dimensions that will reward those who come up with better ways to program. At PARC we changed Smalltalk every few weeks, and in a major way every two years. Though it has hardly changed since then, please do and put those big changes out on the Internet for all of us to learn from and enjoy!

About the Author

■ **STÉPHANE DUCASSE** obtained his Ph.D. at the University of Nice-Sophia Antipolis and his habilitation at the University of Paris 6. He was recipient of the SNF 2002 Professeur Boursier Award. He is now Professor at the University of Berne and the Université de Savoie.

Stéphane's fields of interests are design of reflective systems, object-oriented languages design, composition of software components, design and implementation of applications, reengineering of object-oriented applications, and teaching novices. He is the main developer of the Moose reengineering environment. He loves programming in Smalltalk and is the president of the European Smalltalk User Group.

Stéphane has written several books in French and English: *La programmation: une approche fonctionnelle et recursive en Scheme* (Eyrolles 96), *Squeak* (Eyrolles 2001), and *Object-Oriented Reengineering Patterns* (MKP 2002).

If you want to discover why Stéphane is having fun with Squeak and actively participating in its development, check out <http://www.squeak.org/>. Check out <http://smallwiki.unibe.ch/botsinc/> for the web site of this book.

Acknowledgments

I would like to thank all of you who read parts and drafts of this book and provided feedback. It is not an easy task to read a work in progress, and I am grateful to all of you who made the effort. I will not attempt to list all your names here, because I am sure to forget some of you. However, I must mention Orla Greevy, Ian Prince, and Daniel Knierim, who read the entire manuscript. Thank you for your feedback and support. I would also like particularly to mention Daniel Villain, who read a draft of the French version.

I want to thank the Squeak community for the help they have provided me during the development of the environments used in this book, and for developing the amazing Squeak environment in the first place. In particular, I would like to thank Nathanael Schärli and Ned Konz for their help. I offer special thanks to all the developers who helped Smalltalk to escape from the clouds of dreamland and become a reality. I would also like to thank all the “Smalltalkers” who made this language and community so exciting. May you continue to make your dreams come true.

Writing this book has been a long and difficult process, because teaching novices is difficult. Moreover, I am not an easy person to live with, and as a researcher, I become excited by too many topics. I want to thank Didier Besset particularly for many fruitful discussions at the beginning of this project.

I also want to thank my wife, Florence, and my sons Quentin and Thibaut, two small boys who loved to run noisily around my desk when I was trying to concentrate on my work. Thank you for accepting a husband and father who was not always present, enthusiastic, and accessible. But soon we will be programming together.

Preface

Knowledge is only one part of understanding. Genuine understanding comes from hands-on experience.

—S. Papert

Goals and Audience

The goal of this book is to explain elementary programming concepts (such as loops, abstraction, composition, and conditionals) to novices of all ages. I believe that learning by experimenting and solving problems is central to human knowledge acquisition. Therefore, I have presented programming concepts through simple problems such as drawing golden rectangles or simulating animal behavior.

My ultimate goal is to teach you object-oriented programming, because this particular paradigm provides an excellent metaphor for teaching programming. However, object-oriented programming requires some more elementary notions of programming and abstraction. Therefore, I wrote this book to present these basic programming concepts in an elementary programming environment with the special perspective that this book is the first in a series of two books. Nevertheless, this book is completely self-contained and does not require you to read the next one. The second book introduces another small programming environment. It focuses on intermediate-level topics such as finding a path through a maze and drawing fractals. It also acts as a companion book for readers who want to know more and who want to adapt the environment of this book to their own needs. Finally, it introduces object-oriented programming.

The ideal reader I have in mind is an individual who wants to have fun programming. This person may be a teenager or an adult, a schoolteacher, or somebody teaching programming to children in some other organization. Such an individual does not have to be fluent in programming in *any* language.

The material of this book was originally developed for my wife, who is a physics and mathematics teacher in a French school where the students are between eleven and fifteen years old. In late 1998, my wife was asked to teach computing science, and she was dismayed by the lack of appropriate material. She started out teaching HTML, Word, and other topics, and she remained dissatisfied, since these approaches failed to promote a scientific attitude toward computing *science*. Her goal was to teach computer science as a process of attacking problems and finding solutions.

As a computer scientist, I was aware of work on the programming language Logo, and I particularly liked the idea of experimentation as a basis for learning. I was also aware that the programming language Smalltalk had been influenced by the ideas of Seymour Papert and those behind Logo, and that it had originated from research on teaching programming to children. Moreover, Smalltalk has a simple syntax that mimics natural language. At about that time, the Squeak environment had arrived at a mature state, and books started to become available in late 1999. But these were for experienced programmers, so I started and wrote the present book.

The environments that I use in this book and its companion book are fully functional. They have gone through several iterations of improvements based on the feedback that I have received from teachers. A guiding rule in our work has been to modify the Squeak environment as little as possible, for our goal is for readers to be able to extend the ideas presented in this book and develop new ones of their own.

Object-Oriented Structure and Vocabulary

The chapters of this book are relatively small, so that each chapter can be turned into a one- or two-hour lab session. I do not advocate presenting the material directly to children for self-instruction, but each chapter in fact has all the material for such an approach.

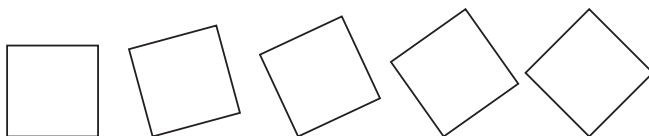
Although object-oriented programming is not developed in this book, I use its vocabulary. That is, we create objects from classes and send them messages. Object behavior is defined by methods. I made this choice because the metaphor offered by object-oriented programming is natural, and children have an intuitive understanding of the idea of objects and their behavior.

Those who are used to Logo may wonder why our robots do not have “pen up” and “pen down” methods, but instead “go” and “jump,” where under the former, a robot moves leaving a trace, while the latter moves a robot forward without leaving a trace. I believe that the go and jump paradigm is better suited to the ideas of object-oriented programming and encapsulation of data than the traditional pen down and pen up design. An excellent analysis of these two approaches was made by Didier Besset, who collaborated with me on this project in its early stages.

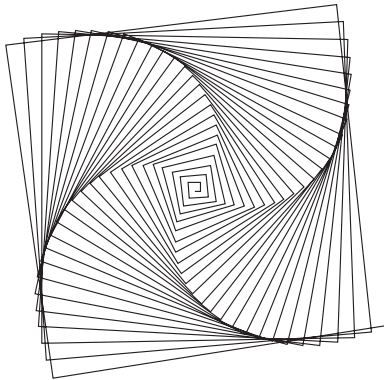
Organization

The book is divided into five parts, as described below.

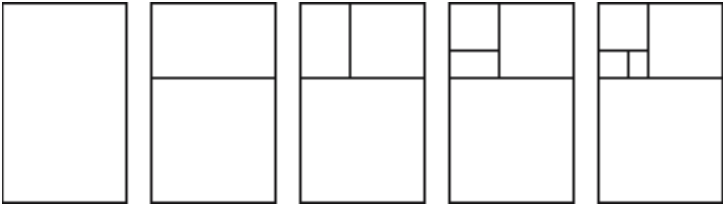
Getting Started. Part 1 shows how to get started with the Squeak environment. It explains the installation process and how to launch Squeak, and then presents robots and their behavior. A first simple program that draws some lines is presented.



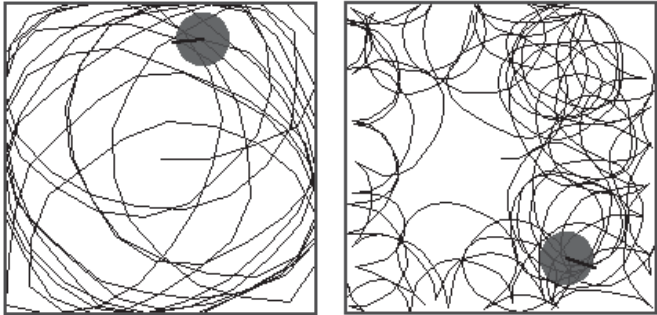
Elementary Programming Concepts. Part 2 introduces first programming concepts, such as loops and variables. It shows how messages sent to a robot are resolved.



Bringing Abstraction into Play. Part 3 introduces the necessity of abstraction, that is, methods or procedures that can be reused by different programs. The most difficult concept introduced is the idea of composing new methods from existing ones to solve more complex problems. Several nontrivial experiments are proposed, such as drawing golden rectangles. Techniques and tools for debugging programs are also introduced.



Conditionals. Part 4 introduces the notion of conditionals, conditional loops, and Boolean expressions, all of which are central to programming. This part also introduces the notion of references in a two-dimensional space and some other types of robot behavior. Finally, ways of using robots to simulate the behavior of simple animals are presented.



Other Squeak Worlds. Part 5 presents two entertaining programming environments that are available in Squeak: the eToy graphical scripting system and the 3D authoring environment Alice.

Why Squeak and Smalltalk?

You may be wondering why among the large number of programming languages available today I have chosen Smalltalk. Smalltalk and Squeak have been chosen for the following reasons:

- Smalltalk is a powerful language. You can build extremely complex systems within a language that is simple and uniform.
- Smalltalk was designed as a teaching language. It was influenced by Logo and LISP, and Smalltalk in turn heavily influenced languages such as Java and C#. However, those languages are much too complex for a first exposure to programming. They have lost the beauty of Smalltalk's simplicity.
- Smalltalk is dynamically typed, and this makes transparent a number of concerns related to types and type coercion that are tedious to explain and of little interest to the novice.
- With Smalltalk you need to learn only key, essential concepts, concepts that are to be found in all programming languages. Thus with Smalltalk I can focus on explaining the important concepts without having to deal with difficult or unattractive aspects of more complex languages.
- Squeak is a powerful multimedia environment, so after reading my books you will be able to build your own programs in a truly rich context.
- Squeak is available without charge and runs on all of today's principal computing platforms. And it should be easily portable to the platforms of the future.
- Squeak is popular. For example, in Spain, it is used in schools, where it runs on over 80,000 computers.

PART 1



Getting Started



Installation and Creating a Robot

Set your stopwatch! Five minutes from now, the robot playground, called the *environment*, that you will be using in this book will be up and running and ready for you to have fun in. In this chapter you will learn how to install the environment, become acquainted with its different parts, and begin interacting with the robots that live in this environment. You will learn how to program these robots to accomplish challenging tasks by sending them *messages*.

So let us get started installing the environment and preparing for all the challenges ahead in the rest of the book. If your environment is already installed, then turn off your stopwatch, skip the first section, and plunge directly into the following sections, which give an overview of the environment. After you have acquired some facility with robots in Chapters 2 through 4, I will go into more detail on using the environment in Chapter 5.

Installing the Environment

The environment used in this book has been developed to run on top of Squeak. Squeak is a rich and powerful Open Source multimedia environment written entirely in Smalltalk and freely available for most computer operating systems at <http://www.squeak.org>. Note, however, that you will not be using the default Squeak distribution. Rather, you will be using a distribution that I have prepared for use with this book. It can be downloaded from the publisher of this book at <http://www.apress.com>, in the Downloads section.

Squeak runs exactly the same on all platforms. However, to make your life a little easier, I have prepared several platform-dependent compressed files. The principle is exactly the same on a Mac, PC, or any other platform. The only differences are in the tools for file decompression and the way that you will invoke Squeak. Once you have obtained a file named `ReadyToUse.zip`, you decompress it and then drag the file named `Ready.image` (Mac) or `Ready(PC)` onto the Squeak application, and that does it! The file `Ready[.image]` contains the complete environment used in this book. Note that you may get files with slightly different names, but that should have no effect on how everything works.

Installation on a Macintosh

For installation on a Macintosh, you should have a ZIP archive file named `readyToUse.zip`. Normally, double clicking on the file's icon should invoke the proper decompression software, such as StuffIt Expander. Once this archive has been decompressed, you should end up with four files, as shown in Figure 1-1. You should identify two files: the file named `Ready.image` and the *Squeak application* file (the one without a file extension in Figure 1-1; it is named Squeak).

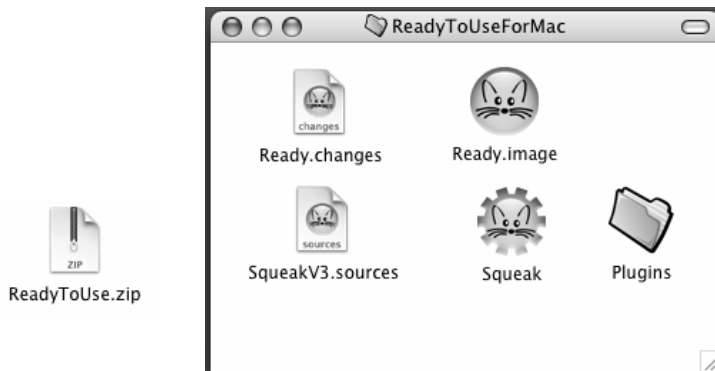


Figure 1-1. Ready-to-use files for the Macintosh. Left: the ZIP archive. Right: the decompressed files.

Installation under Windows

For installation under Windows, you should have a ZIP archive file named `readyToUse.zip`. Once this archive has been decompressed using WinZip, you should end up with four files, as shown in Figure 1-2. You should identify two files: the file named `Ready` and the *Squeak application* file (the one without a file extension in Figure 1-2; it is named Squeak).

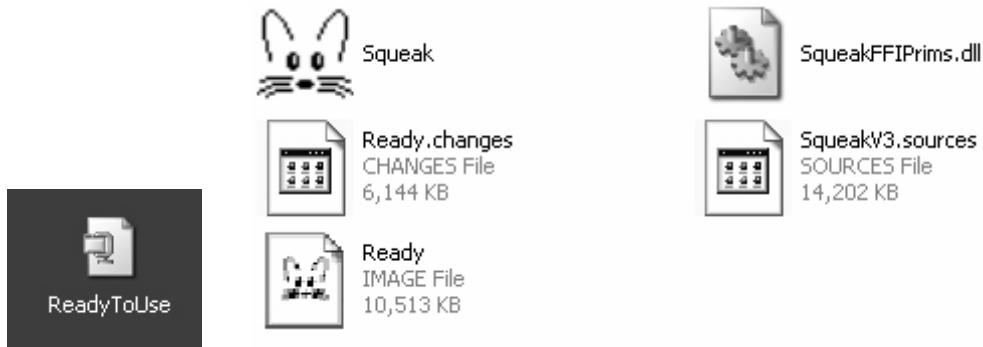


Figure 1-2. Ready-to-use files for Windows. Left: the ZIP archive. Right: the decompressed files.

Opening the Environment

To open the environment, drag the file Ready[.image] onto the *Squeak application*, that is, onto the file named Squeak, as shown in Figure 1-3. You should obtain the environment shown in Figure 1-4. If you do not get this environment, then read the section “Installation Troubleshooting” near the end of this chapter.

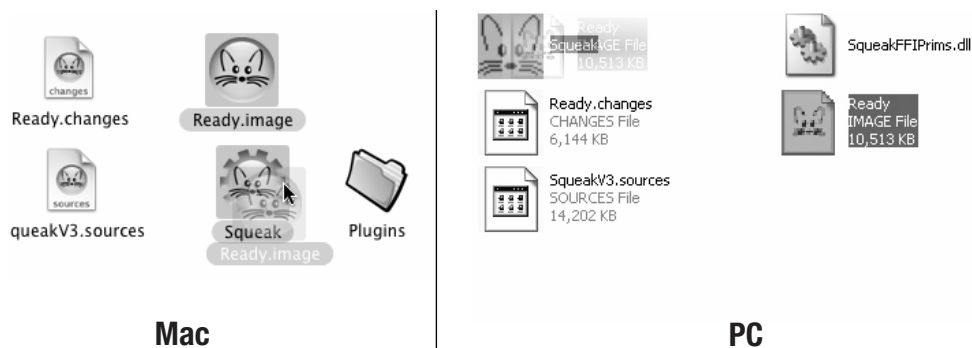


Figure 1-3. Dragging and dropping the image file onto the Squeak application file opens the environment on a Mac (left) or on a PC (right).

Tips for Installation

The environment can be opened simply by double clicking on the image file. However, there are several disadvantages to this: You may have to identify the Squeak application, and sometimes another application may interfere and try to use the image file. Moreover, you can find yourself in trouble if you have multiple installations of different versions of Squeak. So I suggest that you always open the environment by dragging and dropping the image file onto the Squeak application file or an alias of it.

Note that if you do not have enough space for the installation on your hard drive, you can use an alias to the SqueakV3.sources file, which can be shared among several installations.

Important! To start the environment, drag and drop the file `Ready` (with the `.image` extension for Mac) onto the Squeak application.

First Interactions with a Robot

Once you have opened the environment by dragging the file named `Ready` [`.image`] onto the Squeak application as explained previously, the environment that you obtain should look something like the one presented in Figure 1-4.

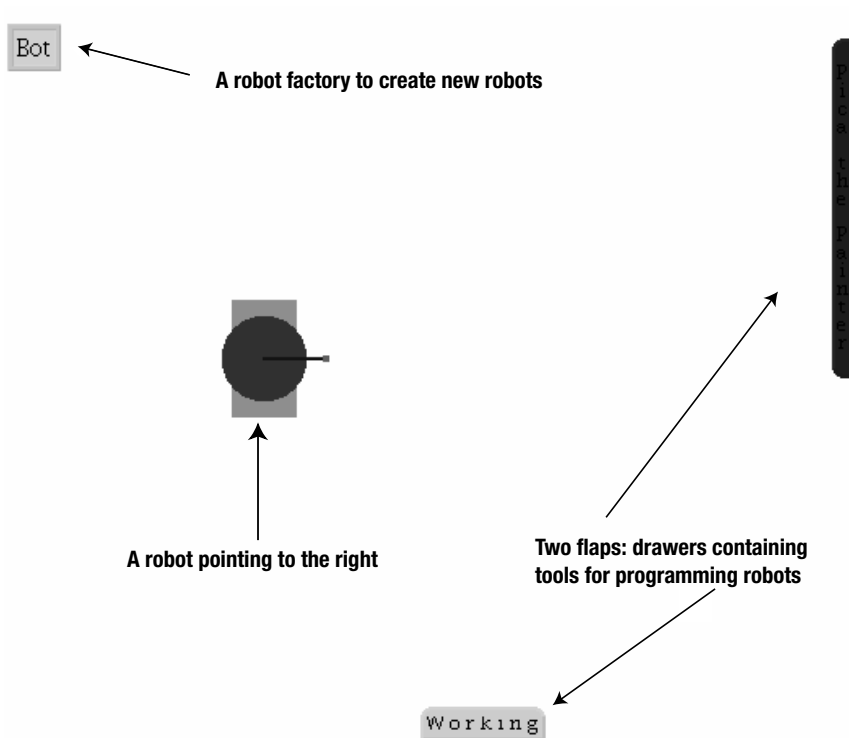


Figure 1-4. *The environment is ready to use.*

The environment is composed of a robot factory and two flaps. A flap is a drawer containing programming tools. You will not need these for a while, and so I will put off describing them until a later chapter. You should see a small blue robot in the middle of the screen. This is not a robot made of wires and metal, but a software robot, imagined as seen from above, pointing toward the right edge of the screen. A robot is a round blue circle; it has two wheels and a small red head that points in its current direction. As you work through this book, you will be sending orders to robots. These orders are called *messages*, and we say that the robots *execute* these messages.

Place the mouse over the robot and wait a second. A balloon pops up with some information about the robot, such as its current location and its direction, as shown in Figure 1-5. Since computer monitors are of varying sizes and resolutions, your robot's position may have other values.

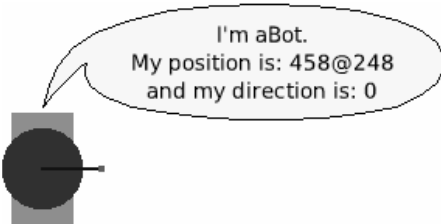


Figure 1-5. Place the mouse over a robot to pop up a balloon with information about the robot.

Sending Messages to a Robot

You can interact directly with a robot by left clicking on the robot with the mouse (or just clicking with a one-button mouse). A messaging balloon pops up, as shown in the left picture in Figure 1-6. In this balloon you can type messages to be sent to the robot. After you type your messages, you send them to the robot by pressing the return key, and the robot then executes them.

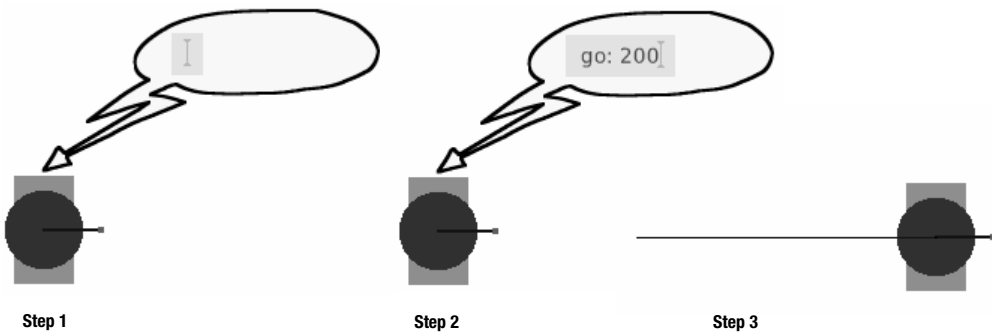


Figure 1-6. Step 1: Left-clicking on a robot causes a messaging balloon to appear. Step 2: You can type a message to the robot to move 200 pixels forward and then press the return key. Step 3: The robot executes the message; it has moved, leaving a trace on the screen behind it.

For example, if you type the message `go: 200` followed by the return key, you have told the robot to move forward 200 pixels in its current direction. If you type the message `turnLeft: 20 + 70`, you are instructing the robot to turn to its left (counterclockwise) $20 + 70 = 90$ degrees, as shown in Figure 1-7. This second message is more complex than the previous one, because the value representing the number of degrees that the robot is to turn is itself a message (as I will soon explain), namely, `20 + 70`. We will call such messages *compound messages*.

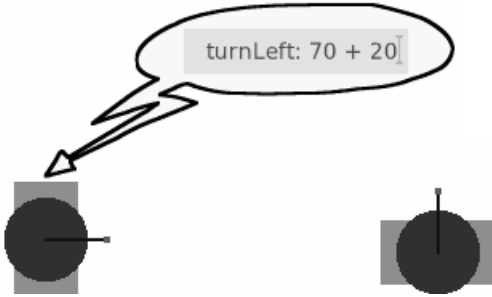


Figure 1-7. *Left: sending a compound message. Right: The message has caused the robot to turn to its left by 90 degrees.*

When the message `color: Color green` is sent to a robot, it changes its color, as shown in Figure 1-8. (You will have to imagine the green color in the grayscale picture.)

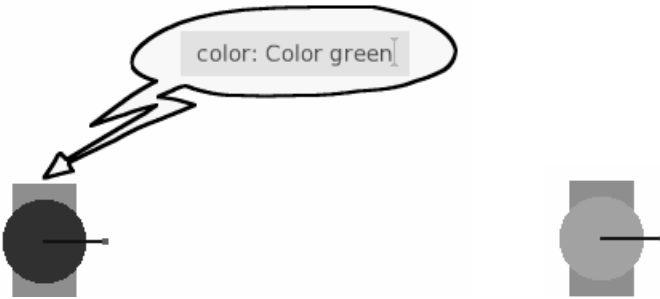


Figure 1-8. *Left: The robot is instructed to change its color to green. Right: The color has changed.*

You may not understand the format of the messages that I have just presented. Some of them may appear a bit complex. In fact, `color: Color green` is another compound message. I will explain later how you can develop your own messages. For now, simply type the messages presented to you so that you can become familiar with the robot's environment. If you want to repeat a previous message, you do not have to retype it. Simply use the up and down arrows to navigate over the previous messages that you have sent to the robot. In subsequent chapters, you will learn step by step all the messages that a robot understands, and what is more, you will learn how to define new behaviors for your robots.

Note To interact with a robot, click on it, type a message, and press the return key.

Creating a New Robot

The environment already contains a robot, but now I am going to show you how to create new robots. If you are not satisfied with having only one robot, you can create a new one by sending the appropriate message to a robot *factory*. A robot factory is graphically represented as an orange box surrounded by a light blue box, in the middle of which the word Bot is written, as shown in Figure 1-9. In Squeak jargon, and in general in the jargon of object-oriented programming, a robot factory is called a *class*. Classes (factories that produce *objects*, such as robots) have a name starting with an uppercase letter. Hence this is the class Bot and not bot.

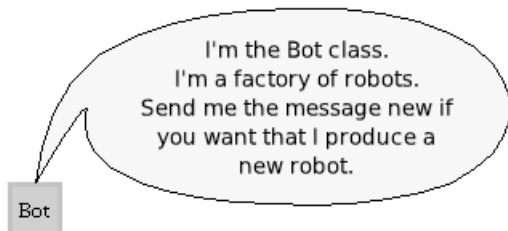


Figure 1-9. In Squeak jargon, a robot factory is called a class. Classes produce objects. The Bot class produces new robots.

Just as you did for robots, you can interact with a robot factory by sending it messages. The message to create a new robot is the message `new`, as shown in Figure 1-10. Note that newly created robots, like your original robot, point to the right of the screen. Each of the two robots has an independent existence, and you can send messages to each of them in turn.

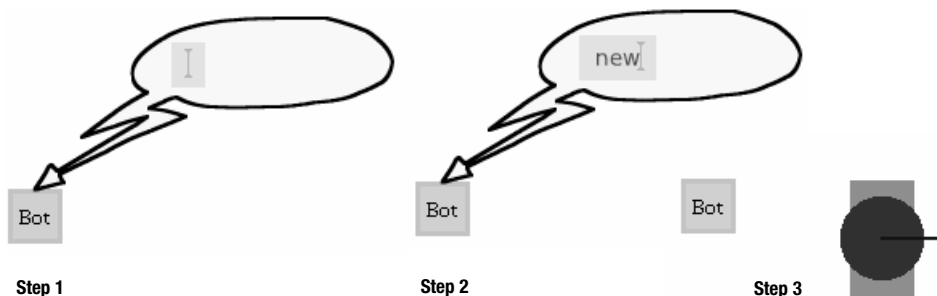


Figure 1-10. Step 1: Start typing a message. Step 2: The message `new` has been sent to the robot factory. Step 3: In response, the factory has created a robot and delivered it to you.

To create a new robot, send the message `new` to the robot factory, which is the class Bot. When a robot is created, it is always pointing to the east, that is, to the right of the screen.

Quitting and Saving

The background of the Squeak window application is called the World. The World has a menu offering a number of different options. To display the World menu, just (left) click on the background. You should get a menu similar to the one shown in Figure 1-11. The last group of options consists of all the actions that you can take to quit out of the environment or save your work.

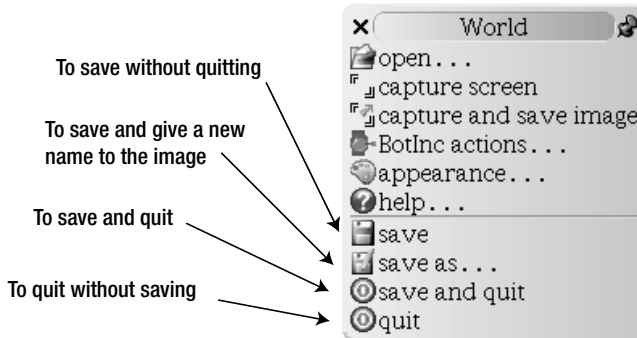


Figure 1-11. *The World menu includes actions for quitting and saving.*

Selecting the item **quit** simply quits the environment without saving your work. The result is that the next time you launch the environment, it will be in exactly the same state as the last time you saved it. Selecting the item **save** saves the complete environment. The next time you start the environment, it will be in exactly the same state as the last time you saved it. Finally, if you select the item **save as...**, the environment asks you to create a new name, and it will then create two new files with that name: one with the extension `.image` and one with the extension `.changes`. That is how I created the files `Ready[.image]` and `Ready.changes`. To open the environment that you saved with a new name, drag and drop the file with the new name that has the extension `.image` onto the squeak application file icon as you did to start the environment by dragging and dropping the file `Ready[.image]`.

Installation Troubleshooting

Sometimes things don't proceed just as they should, so in this section I will present some information that should be of help if you encounter problems during installation. First, I will explain the role of the principal files that you obtained when you decompressed the archive.

To run the environment provided with this book or with any Squeak distribution, four files are necessary. Knowing about them can help in solving any problems you may encounter.

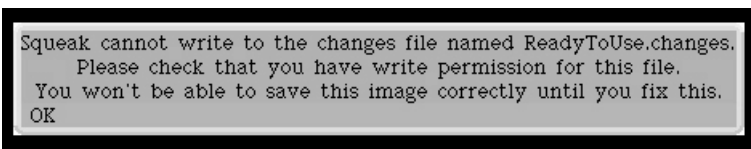
Image and changes. The file `Ready[.image]`, called simply the *image* file, and the file `Ready.changes`, called simply the *changes* file, contain information about your current Squeak system. These two files are synchronized by Squeak automatically and should be writable (that is, not read-only). Each time you save your environment, these two files are

synchronized. You should not edit them with a file editor or change the name of the file manually. If you want to use different names, just use the **save as...** menu item of the World menu. Squeak will then create a new pair of files for you.

Source. The file named `SqueakV3.sources`, called the *sources* file, contains the source code of a part of the Squeak environment. You will not need it in working through this book, so do not try to edit it manually. However, this file should always be in the same directory in which the image file is located.

Application. The application files `Squeak` for Mac and `Squeak.exe` for PC are the Squeak application. Each of these files is the application that runs when you are programming in Squeak. It should be executable. This file is referred to as the *Squeak application*. In computer-science jargon, this application is called a *virtual machine*, or VM for short.

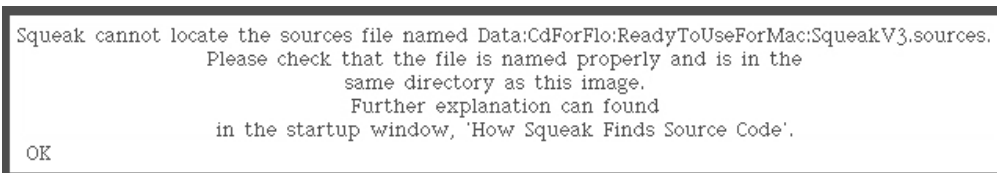
Keep in mind that the image and changes files should be writable. Some operating systems change the properties of files to “read only” when they are copied from an external source. If that happens, Squeak warns you with a message, like that shown in Figure 1-12. If you get such a message, simply quit Squeak without saving, change the property of the file to permit write access, and restart.

A screenshot of a Squeak error message dialog box. The text inside the dialog box reads: "Squeak cannot write to the changes file named ReadyToUse.changes. Please check that you have write permission for this file. You won't be able to save this image correctly until you fix this. OK". The dialog box has a black border and a light gray background.

```
Squeak cannot write to the changes file named ReadyToUse.changes.  
Please check that you have write permission for this file.  
You won't be able to save this image correctly until you fix this.  
OK
```

Figure 1-12. This message appears if the image (`Ready.[image]`) or changes (`Ready.changes`) file is not writable.

Another possible problem you may encounter is related to the sources file `SqueakV3.sources`. This file or an alias pointing to this file should be present in the directory in which the image file is located. If the file itself is not present, you may get the message shown in Figure 1-13. To cure this problem, create an alias to the sources file (`SqueakV3.sources`) in the directory containing the image file or simply copy the sources file into the directory that contains the image file. You should not have this problem if you are using the distribution for this book.

A screenshot of a Squeak error message dialog box. The text inside the dialog box reads: "Squeak cannot locate the sources file named Data:CdForFlo:ReadyToUseForMac:SqueakV3.sources. Please check that the file is named properly and is in the same directory as this image. Further explanation can found in the startup window, 'How Squeak Finds Source Code'. OK". The dialog box has a black border and a light gray background.

```
Squeak cannot locate the sources file named Data:CdForFlo:ReadyToUseForMac:SqueakV3.sources.  
Please check that the file is named properly and is in the  
same directory as this image.  
Further explanation can found  
in the startup window, 'How Squeak Finds Source Code'.  
OK
```

Figure 1-13. Possible messages indicating that the sources file (`SqueakV3.sources`) is missing from the directory containing the image file.

Summary

To start the environment, drag and drop the file `Ready[.image]` or another file that you have saved with the `.image` extension into the squeak application.

- To send a message to a robot, left click on it, type the message, and press the return key.
- To create a new robot, send the message `new` to the class `Bot`, which is your robot factory.
- When a robot is created, it is always pointing to the east, that is, to the right of the screen.
- To obtain the menu for saving the environment, click on the background.



A First Script and Its Implications

While sending messages using direct interaction with a robot is a fun and powerful way of programming robots, it is rather limited as a technique for writing complex programs. To expand your programming horizons, I am going to teach you about the notion of a *script*, which is a sequence of *expressions*, together with all the fundamental concepts and vocabulary that you will need for the remainder of this book. It also serves as a map to subsequent chapters, which will introduce in depth the concepts briefly presented in this chapter.

First, I will show you how you to send multiple messages to the same robot by separating a sequence of messages with semicolons. Then you will learn how to write a script using a dedicated tool called a *workspace*. I will describe the different elements that compose a script and show some of the errors that one can make when writing a program.

Using a Cascade to Send Multiple Messages

Suppose you want to get your robot on the screen to draw a rectangle of height 200 pixels and width 100 pixels. To do so, you might click on your robot and then start to type the first message, `go: 100`, press the return key, then click on the robot and type the second expression, `turnLeft: 90`, and press the return key, then click on the robot and type the expression `go: 200`, and so on. You will quickly notice that this is truly a tedious way of interacting with your robot. It would be much more convenient if you could first type in all the instructions and then push a button to have the sequence of instructions executed.

In fact, you can send multiple messages to a robot by separating the messages with a semicolon character `;`. To send a robot the messages `go: 100`, `turnLeft: 90`, and `go: 200`, simply separate them with a semicolon as follows: `go: 100 ; turnLeft: 90 ; go: 200` (see Figure 2-1). This way of sending multiple messages to the same robot is called a *cascade of messages* in Squeak jargon.

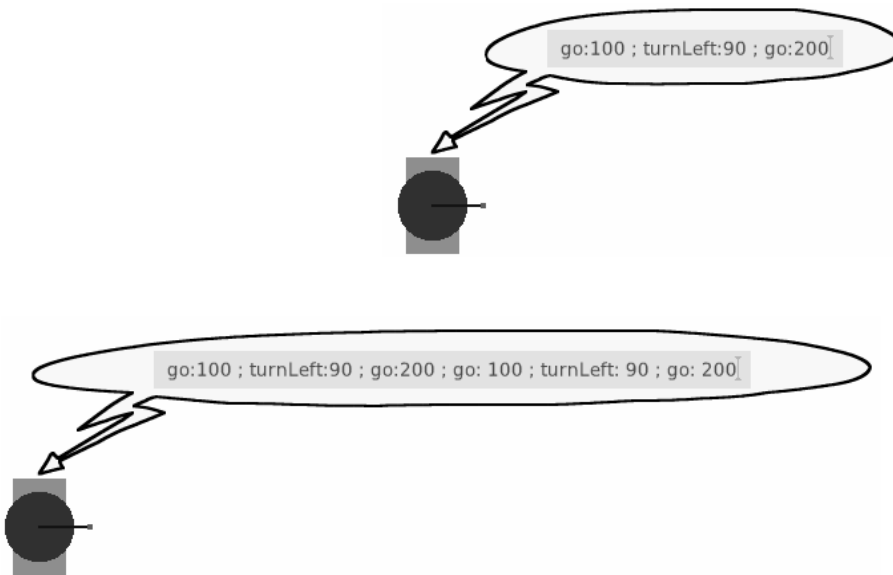


Figure 2-1. You can send several messages to a robot at once using the semicolon character.

However, the technique of writing a cascade of messages (that is, sending a robot multiple messages separated by semicolons) does not work well for complex programs. Indeed, even for drawing a simple rectangle, the string of messages quickly grows too long, as shown by the second message in Figure 2-1. And there are other concerns as well. For example, programmers take into account issues such as whether they can store a sequence of messages and replay them later and whether they can reuse their messages and not have to type them in all the time. For all these reasons, we need other ways to program robots. The first way that you will learn is to write down a sequence of messages, called a *script*, in a text editor and ask the environment to execute your script.

A First Script

The BotInc environment provides a small text editor, called the Bot Workspace, which is dedicated to script execution (that is, executing the expressions that constitute a script). Click on the bottom flap, called *Working*. By default, it contains a Bot Workspace editor, as shown in Figure 2-2.

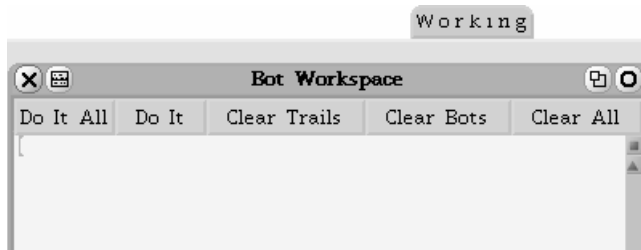


Figure 2-2. A Bot workspace is a small text editor dedicated to the execution of robot scripts.

I will start off by writing a script that draws a rectangle, and then I will explain it in detail (Script 2-1).

Script 2-1. *The robot pica is created and is made to move and turn.*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90
```

Figure 2-3 shows the script in a Bot workspace and the result of its execution obtained by pressing the **Do It All** button. Try to get the same result: type the script and press the **Do It All** button. I have named the robot *pica* as short for Picasso, since our robots are drawing pictures, just like those of the great Spanish artist.

The **Do It All** button of the Bot workspace executes *all* the messages that the workspace contains. Therefore, before typing a script, make sure that no other text is already present in the Bot workspace. Moreover, computers and programming languages cannot deal with even the most obvious mistakes, so be careful to type the text exactly as it is presented in Script 2-1. For example, you must type the uppercase “B” of Bot on the second line, and you must end each line with a period. (There is no need to put a period at the end of the last line, because periods *separate* messages in Squeak. There is also no need for a period after the first line, because it doesn’t contain a message.) But more on that a bit later in the chapter. The script and its result are shown in Figure 2-3.

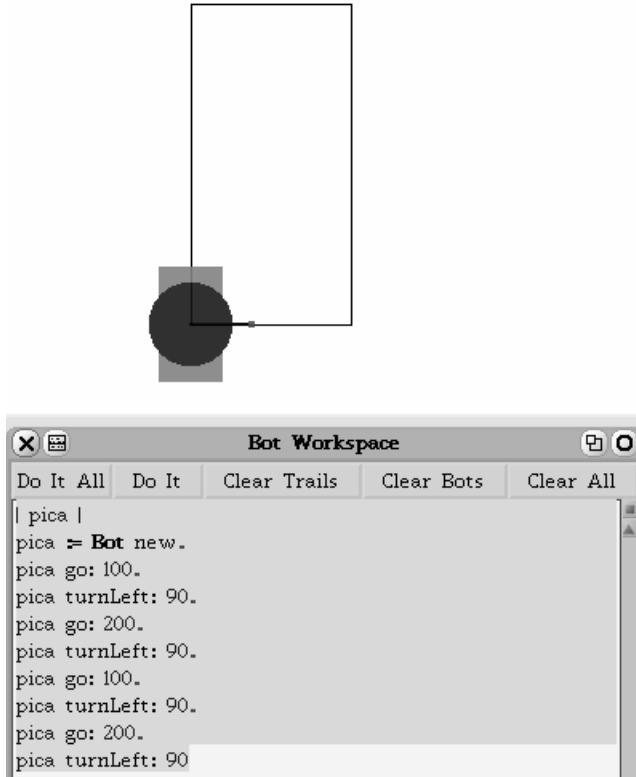


Figure 2-3. A script executed using the *Do It All* button of the *Bot workspace* and its result.

Squeak and Smalltalk

Script 2-1 is admittedly simple, but nonetheless, it constitutes a genuine computer program. A *program* is a list of *expressions* that a computer can execute. To define programs we need programming languages, that is, languages that allow programmers to write instructions that a computer can “understand” and execute.

Programming Languages

A well-designed programming language serves to support programmers in expressing solutions to their problems. By support, I mean that the language should, among other things, facilitate expression of the task to be performed, provide efficient execution of the program code and reliability of the resulting application, give the programmer the ability to prove that programs are correct, encourage the production of readable code, and make it easy for programmers to make changes in their applications. There is no “best” or ideal programming language that satisfies all of these desirable properties, and different programming languages are best suited for different kinds of tasks.

Smalltalk and Squeak

This book will teach you how to program in the Smalltalk *programming language* within the Squeak *programming environment*. A programming environment is a set of tools that programmers use to develop applications. Squeak contains a large number of useful tools: text editors, code browsers, a debugger, an object inspector, a compiler, widgets, and many others. And that's not all! In the Squeak environment you can program music, animate flash files, access the Internet, display 3D objects, and much more. However, before you can start programming complex applications, you have to learn some basic principles, and that is the purpose of this book.

Squeak programmers develop their applications by writing programs using the programming language called Smalltalk. Smalltalk is an *object-oriented* programming language. Other object-oriented programming languages are Java and C++, but Smalltalk is the purest and simplest. As the term “object-oriented” suggests, such programming languages make use of *objects*. The objects that are created and used are, of course, not real objects, but logical structures, or “virtual” objects, within the computer. But they are called objects because it is useful to think of these structures as manufactured contraptions, such as a robot, for example, that are able to understand messages that are sent to them and to execute whatever instructions are contained in those messages. The point of the object analogy is that we can use a robot, or a radio, or a camera, without understanding its internal structure. We need only know how to use it by pushing its buttons or sending it messages via the remote control.

Where do manufactured objects come from? A factory, of course. The factories used to create objects are called *classes* in object-oriented programming languages. Defining classes is somewhat tricky, as is object-oriented programming in general, so in this introductory book I will not show you how to define classes. Instead, you will only define new types of behavior for your robot, and this will give you a good grounding in basic programming concepts.

I chose Smalltalk as the language for this book because it is simple, uniform, and pure. It is pure in that in Smalltalk, *everything* is an object that sends and receives messages to and from other objects. It is simple because in Smalltalk there are only a few basic rules, and it is uniform in that these rules are always applied consistently. In fact, Smalltalk was originally designed for teaching novices how to program. But that doesn't mean that Smalltalk can be used only for writing “baby” or “toy” applications. Indeed, large and complex applications have been written in Smalltalk, such as the applications controlling the machines that produce the AMD corporation's microprocessors that may be running in your computer.

Another application written entirely in Smalltalk is the Squeak environment itself. Now isn't that interesting! This means that once you develop a good understanding of Smalltalk, you can modify the Squeak environment in order to adapt the system to your own purposes or simply to learn more about the system. With Smalltalk, then, you have quite a bit of power in your hands.

I hope that this discussion about programming languages in general, and Smalltalk in particular, has motivated you to learn how to program. But please be aware that learning to program is like learning to play the piano or to paint in oils. It is not simple, and so do not become discouraged if you have some difficulties. Just as a beginning piano player doesn't start off with Beethoven's *Waldstein Sonata*, and a novice painting student doesn't try to reproduce Michelangelo's Sistine Chapel ceiling, the beginning programmer starts off with simple tasks. I have designed this book so that topics are introduced in a logical order, so that what you learn in each chapter builds on your knowledge from previous chapters and prepares you for the material in the following chapters.

Programs, Expressions, and Messages

Now we are ready to take a closer look at your first script and explain just what is going on.

Typing and Executing Programs

When you wrote Script 2-1, you typed some text, constituting a sequence of expressions, and then you asked Squeak to execute it by pressing the **Do it All** button. Squeak executed the sequence of expressions; that is, it transformed the textual representation of your program into a form that is understandable by a computer, and then each expression was executed in sequence. In this first script, executing the sequence of expressions created a robot named *pica*, and then *pica* executed, one after another, the messages that were sent to it.

A program in Squeak consists of a sequence of *expressions* that are executed by the Squeak environment. In this book, such a sequence is called a *script*.

Important! A *script* is a sequence of expressions.

A program is a bit like a recipe for a chocolate cake. A good cake recipe describes all the steps to be carried out in correct sequence: cream the butter and sugar; melt the chocolate; add the chocolate to the butter and sugar mixture; sift in the flour; and so on right through placing the filled cake pans in a 350° oven, cooling the baked cake on a rack, and spreading on the frosting. Enjoy! Similarly, a computer program describes all the steps in sequence needed to produce a certain effect: declare a name for a robot; create a robot with that name; tell the robot to move 100 pixels; tell the robot to turn; and so on.

The Anatomy of a Script

The time has arrived to analyze your first script, which is copied here as Script 2-2.

Script 2-2

```
| pica |  
pica := Bot new.  
pica go: 100.  
pica turnLeft: 90.  
pica go: 200.  
pica turnLeft: 90.  
pica go: 100.  
pica turnLeft: 90.  
pica go: 200.  
pica turnLeft: 90
```

In a nutshell, Script 2-2 starts off by declaring that it will be using a *variable* named *pica* to refer to the robot it creates. Once the robot is created and associated with the variable *pica*, the script tells the robot to take a sequence of walks to different locations on the screen while

turning 90 degrees to the left after each walk. Now let us analyze each line step by step. Don't worry if certain concepts such as the notion of a variable remain a bit fuzzy. Everything will be dealt with in due course, and if not in this chapter, then in a future chapter.

| `pica` | This first line declares a *variable*. It tells Squeak that we want to use the name `pica` to refer to an object. Think of it as saying to a friend, from now on I am going to use the word `pica` in my sentences to refer to the robot that I am about to order from the robot factory. You will learn more about variables in Chapter 8.

`pica := Bot new.` This line creates a new robot by sending the message `new` to the robot factory (class) named `Bot` and associates the robot with the name `pica`, the variable that was declared in the previous step. The word `Bot` requires an uppercase letter B because it is a class, in this case the class that is a factory for producing robots.

`pica go: 100.` In this expression, the message `go: 100` is sent to the robot we named `pica`. This line can be understood as follows: “`pica`, move 100 units across the computer monitor.” It is implicit in this expression that a robot receiving a `go: message` knows in what direction to travel. In fact, a robot is always pointing in some direction, and when it receives a `go: message`, it knows to move in the direction in which it happens to be pointing. Note also that the message name `go:` terminates with a colon. This indicates that this message needs additional information, in this case a length. For example, `go: 100` says that the robot should move 100 pixels. The message name is `go:`.

`pica turnLeft: 90.` This line tells `pica` to turn 90 degrees to its left (counterclockwise). This line is again a message sent to the robot named `pica`. The message name `turnLeft:` ends with a colon, so additional information is required, this time an angle.

The remaining lines of the script are similar.

Important! Any message name that terminates with a colon indicates that the message needs additional information, such as a length or an angle. For example, the message name `turnLeft:` requires a number representing the angle through which the robot is to turn counterclockwise.

About Pixels

On a computer screen, the unit of distance is called a *pixel*. This word was invented in about 1970 and is short for “picture element.” A pixel is the size of the smallest point that can be drawn on a computer screen. Depending on the type of computer monitor you are using, the actual size of a pixel can vary. You can see individual pixels by looking at the screen through a magnifying glass.

Expressions, Messages, and Methods

I have been using the terms *expression* and *message*. And now it is time to define them. I will also define the important term *method*.

Expression

An *expression* is any meaningful element of a program. Here are some examples of expressions:

- `| pica |` is an expression that declares a variable (more in Chapter 8).
- `pica := Bot new` is an expression involving an operation, called *assignment*, that associates a value with a variable (see Chapter 8). Here, the newly created robot obtained by sending the message `new` to the class `Bot` is associated with the variable `pica`.
- `pica go: 100` is an expression that sends a message to an object. Such an expression is called a *message send*. The message `go: 100` is sent to the object named `pica`.
- `100 + 200` is also a message send. The message `+ 200` is sent to the object `100`.

Message

A *message* is a pair composed of a message name, also called a *message selector*, and possible message *arguments*, which are the values that the object receiving the message needs for executing the message. These relationships are illustrated in Figure 2-4. The object receiving a message is called a *message receiver*. A message together with the message receiver is called a *message send*. Here are some examples of messages:

- In the expression `pica beInvisible`, the message `beInvisible` is sent to a receiver, a robot. This message has no arguments.
- In the expression `pica go: 100`, the message `go: 100` is sent to a receiver, a robot named `pica`. It is composed of the method selector `go:` and a single argument, the number `100`. Here, `100` represents the distance in pixels through which the robot should move. Note that the colon character is part of the message selector.
- In the expression `33 between: 30 and: 50`, the message `between: 30 and: 50` is composed of the method selector `between:and:` and two arguments, `30` and `50`. This message asks the receiver, here the number `33`, whether it is between two values, here the numbers `30` and `50`.
- In the expression `4 timesRepeat: [pica go: 100]`, the message `timesRepeat:` `[pica go: 100]`, which is sent to the number `4`, is composed of the message selector `timesRepeat:` and the argument `[pica go: 100]`. This argument is called a *block*, which is a sequence of expressions (in this case a single expression) inside square brackets (more on this in Chapter 7).
- In the expression `100 + 200`, the message `+ 200` is composed of the method selector `+` and an argument, the number `200`. The receiver is the number `100`.

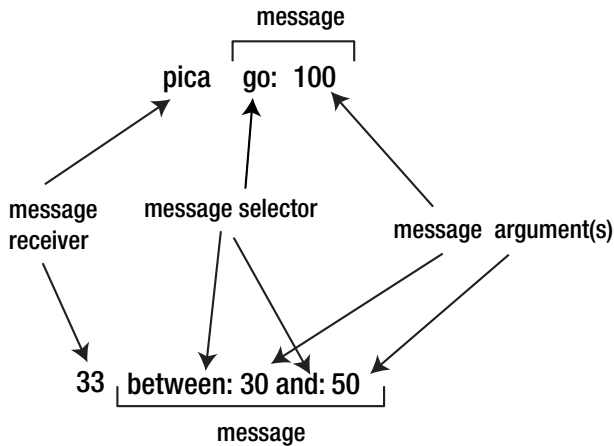


Figure 2-4. Two message sends composed of a message receiver, a message name (or message selector), and a set of arguments

Message Separation

As mentioned earlier, each line of Script 2-1, except the first and last, is terminated by a period. The first line does not contain a message. Such a line is called a *variable declaration* in computer jargon. Thus, we can make the following observation: each message send must be separated from the following one by a period. Note that putting a period after the last message is possible but not mandatory. Smalltalk accepts both.

Important! Message sends should be separated by a period. The last statement does not require a terminal period. Here are four message sends separated by three periods.

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

Important! A period character `.` is a message separator, so there is no need to place one after a message send if there is no following message send. Therefore, no period is necessary at the end of a script or of a block of messages.

Method

When a robot (or other object) receives a message, it executes a *method*, which is a kind of script that has a name. More formally, a method is a named sequence of expressions that a receiving object executes in response to the receipt of a message. A method is executed when an object receives a message of the same name as one of its methods. For example, a robot executes its *method* `go:` when it receives a *message* whose name is `go:`. Thus the expression `pica go: 224` causes the message receiver `pica` to execute its method `go:` with argument `224`, resulting in its moving 224 pixels forward in its current direction. Later in the book, I will explain how you can define new methods for your robot, but for now, we do not need them to start programming.

Cascade

As I mentioned in the first section of this chapter, you can send multiple messages to a robot by separating them with semicolons. Such a sequence of messages is called a *cascade*. You can also use a cascade in a script to send multiple messages to a robot. Script 2-3 is equivalent to Script 2-2, except that now all the messages sent to the robot `pica` are separated by semicolons. Using cascades is handy when you want to avoid typing over and over the name of the receiver of the multiple messages. Cascades are useful because they shorten scripts. However, be careful! Shortcuts can lead to trouble if you don't watch your step, so be sure that you truly intend for all your messages to be sent to one and the same receiver.

Script 2-3

```
| pica |
pica := Bot new.
pica
  go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90 ;
  go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90.
```

Important! To send multiple messages to a robot, use a semicolon character `;` to separate the messages, following the pattern `aBot message1 ; message2`. Here is an example: `pica go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90`

Creating New Robots

To obtain a new robot, you have to send an order to the robot factory to manufacture one for you. That is, you have to send the message `new` to the class `Bot`. There is nothing new here. It is exactly what you did in the previous chapter when you clicked on the blue and orange box named `Bot`, which represents the class of the same name, and typed `new` in the bubble. In Squeak, we always send messages to robots, other objects, or classes to interact with them. There is no difference in treatment, except that classes and objects understand different

messages. It is the job of classes to create objects. An object does not know how to create other objects, so sending a robot the message `new` leads to an error. Classes, on the other hand, generally do not have colors and do not know how to move, and so sending the message `color` or `go: 135` to a class does not make sense, and doing so leads to an error. Nonetheless, in both cases you are sending messages!

The `Bot` class is not the only manufacturing company in the Squeak environment. There are other classes, and they understand different messages and employ different methods for creating different kinds of objects. For example, the class `Color` manufactures color objects. It returns a blue or green color object in response to the message `blue` or `green`. Whenever in this book a new object must be obtained from a specific class, I will tell you how it is done.

Important! To obtain a new object from a class, you generally send the message `new` to the class. Thus `Bot new` creates a new robot. Other classes may offer different messages for obtaining new objects. For example, `Color blue` tells the class `Color` to create a new `blue` color object.

Errors in Programs

Computers are very good at making highly complex calculations at incredible speed, but they lack the intelligence to correct small mistakes. If I accidentally wrote, “now turn on your computer,” you might chuckle over my misspelling, but you would have no trouble understanding what I meant. But computers have no such intelligence, which means that each expression given to a computer must be given precisely, without the least error. The smallest seemingly insignificant mistake in a program, even something as trivial as using a lowercase letter instead of an uppercase one, will almost certainly be misunderstood by the computer. If you have errors in your scripts, two things can go wrong: either an error message will appear on the screen, and this is likely to occur when you are doing your first experiments, or the program will be executed, but the result will not be what you intended. So when things go wrong, do not despair and try to find the error in your program.

Squeak has a helpful error-prevention and error-correction facility. It colors the letters while you are typing. When a word becomes red, this means that you are writing something that Squeak does not understand. An example is shown in Figure 2-5. When a word is blue, for a variable or a message, or black, for a class, this indicates that everything is structurally correct.

If you attempt to execute an expression containing an error, Squeak tries to help you by notifying you when it encounters the error in your code. The error messages that Squeak uses are actually menus. The top part of the menu window contains a short description of the error; then, depending on the type of error, some suggested corrections may be listed as options. If you don't like any of the options, you can always cancel execution by choosing “cancel” in the menu. Then you should locate the place in your script that Squeak did not understand, correct it, and try again to execute the script.

I will now tell you about some of the most common errors.

Misspelling a Message Selector

Misspelling the name of a message leads to an error. In Figure 2-5, I misspelled the message selector `go:`, typing `god:` instead. The message `god:` does not exist in Squeak, and therefore Squeak turned the word red. Ignoring Squeak's friendly warning, I tried to execute the script. Squeak tried to guess what message selector I had in mind, and prompted me with a menu of possibilities. At this point, I could choose the correct message selector (`go:`), and the message `god:` will be replaced by `go:`. Or I can simply choose "cancel." If I take the latter option, I will have to change `god:` to `go:` manually.

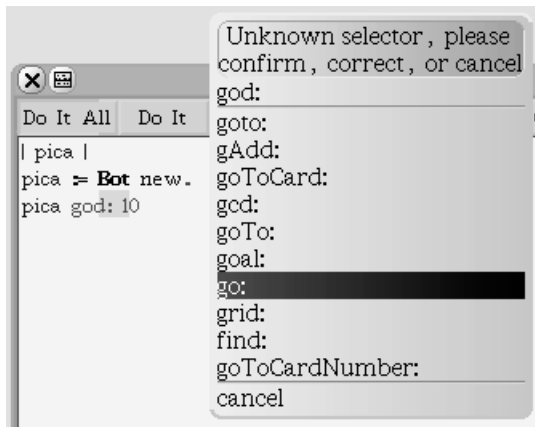


Figure 2-5. I misspelled the message `go:`, typing `god:` instead by mistake. The message `god:` does not exist (in Squeak). Therefore, Squeak prompts you for a possible correction.

Misspelling a Variable Name

There are two ways to misspell the name of a variable: in the body of the script itself and when it is declared (between two vertical bars as in `| pica |`). Figure 2-6 shows the two cases: In the left-hand figure I declared a variable `pica`, but then I typed `pica1` instead of `pica` in the script. Squeak noticed that I was trying to use an undeclared variable, so it turned the text red and prompted me, suggesting that I either declare the undeclared variable by declaring `pica1` as a new variable, or replace `pica1` by `pica`. Since `pica` is the variable name that I wanted, and `pica1` was just a typo, I chose the option `pica`, as shown in the figure. The right-hand figure shows that I accidentally typed a space between the `c` and `a` in `pica` when I attempted to declare the variable `pica`. Squeak did not consider this an error. It simply "thought" that I was trying to declare two variables, `pic` and `a`. Then in the script I typed `pica`, thinking that I had declared that variable. But Squeak saw that in fact, `pica` was an undeclared variable, so it turned the text red and gave me some options, including declaring a new variable with the name `pica` or else replacing what I had typed with the declared variable `pic`.

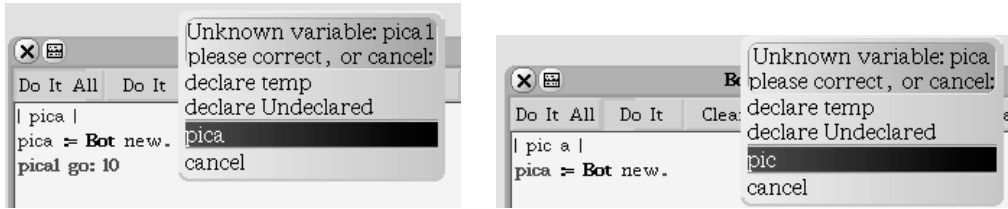


Figure 2-6. Two examples of error. Left: I typed `pica1` instead of the name of the declared variable `pica`. Right: I accidentally typed `pic a` when trying to declare the variable `pica`. This had the result of declaring the variables `pic` and `a` and not a variable called `pica`.

Unused Variables

It may happen that you accidentally declare too many variables. For example, you might declare the variables `pica` and `daly`, thinking that you will need two robots, but then you never use `daly` in your script. This is not really an error, and your program will run correctly even if it has declared variables that are ever used. It is analogous to buying two suitcases, just in case, but using only one of them. You simply have some extra baggage around that you are not using. But just in case you really did mean to use `daly` and forgot, Squeak checks for unused declared variables and if it finds any, suggests that you might want to remove them. For example, in Figure 2-7, the script declares the variables `pica` and `daly` but uses only `pica`. Squeak notices this and asks you whether you would like to remove the unused variable `daly`.

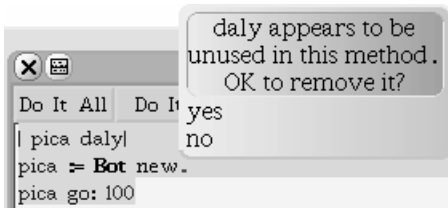


Figure 2-7. All the variables and message sends are correct. However, the variable `daly` is declared but not used, so Squeak indicates this to us and suggests that we might want to remove the unused variable. Unused variables are not an error, but good housekeeping suggests that you should keep things simple and remove them.

Uppercase or Lowercase?

Another common mistake is to forget a required uppercase letter. Names of classes begin with an uppercase letter, so don't forget this when you want to send a message to an object factory. Figure 2-8 shows that I unthinkingly typed `bot` instead of `Bot`. Squeak tried to figure out what I meant, but it failed, and so none of the options that it offered for fixing the problem will do. In such a case you have to correct the error yourself. In the context of this book, the only classes you have to worry about are `Bot`, the robot factory, and `Color`, the color factory.

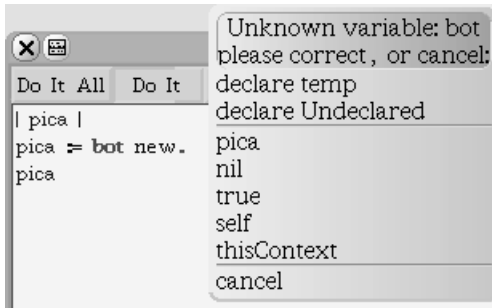


Figure 2-8. I forgot the uppercase B in the name of the class Bot, the robot factory. Squeak knows that something is wrong, but it is not sure what. I will have to correct the error myself.

Forgetting a Period

Finally, one of the most common mistakes, one that even fluent programmers make, is to forget a period between two message sends or a semicolon between two messages in a cascade. A period indicates that a new message send is about to begin, but without the period, Squeak thinks that the current message is being continued, and that the variable meant to be the message receiver of a new message is just another message selector. Since there is no message selector with the name of one of your variables, Squeak tells you that you have typed an unknown selector and offers you some possible corrections. For example, in Figure 2-9, a period is missing after the expression `pica := Bot new`, and Squeak tries to parse (that is, figure out the structure of) the message `pica := Bot new pica go: 120`, and according to the rules of message syntax (structure), about which you will learn in Chapter 11, `pica` should be a message selector. But such a message selector does not exist, so Squeak protests and proposes some possible replacements. Since you know that `pica` is your declared variable and not a message selector, you realize that you forgot a period and so you select “cancel” and type the period manually.

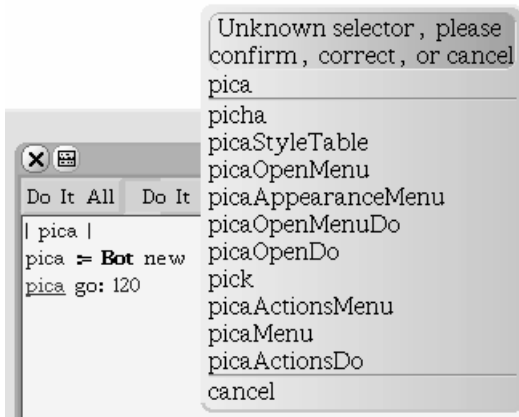


Figure 2-9. The consequences of forgetting a period between message sends: Squeak thinks that the message receiver of the second message send is a nonexistent message selector.

Words That Change Color

Squeak tries to identify mistakes while you are typing your scripts. If it detects something fishy, it changes the color of the text and provides some visual cues that suggest what might be wrong. Figure 2-10 shows some typical situations. Unfortunately, the black-and-white figure does not show its true colors. But use your imagination!

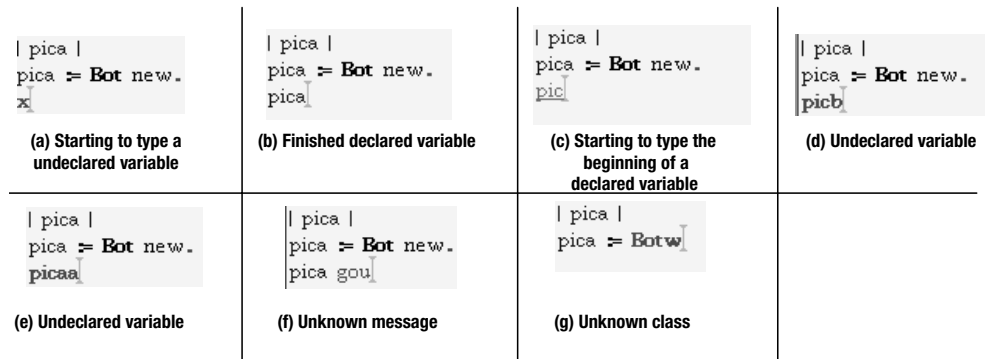


Figure 2-10. *Squeak uses colors to help you find errors and to know when everything is ok.*

Here is a key to the figure:

- (a) I started to type the first letter of an undeclared or unknown variable. Since no variable starting with the letter x has been declared, Squeak turns the x red, letting me know that something is amiss.
- (b) I finished typing a variable that has been declared. Squeak shows me that I have typed a declared variable correctly by turning the text blue.
- (c) I am in the process of typing the name of a variable. As long as what I have typed is the beginning of the name of a declared variable, Squeak underlines it to let me know that so far, everything is ok.
- (d) As soon as I type a character in a variable name that results in a sequence of letters that is not the beginning of the name of a declared variable, Squeak turns the word red. Note the difference with the previous case. In case (c), I could have typed the character a and thereby completed the declared variable pica, as in (a). However, I typed the character b, and ended up with a sequence of letters (picb) that is not the beginning of the name of any declared variable.
- (e) After I typed the name of a declared variable (pica, as in case (b)), I accidentally added an extra character a, which leads to the sequence of letters (picaa), which is not the beginning of the name of a declared variable.
- (f) Squeak tries to do the same for message selectors as it does for variable names. Here I mistyped the message go: and typed instead gou. Squeak was looking for a message selector, and as soon as I typed the character u, it realized that there is no message selector that begins gou, so it turned the text red.

- (g) Squeak tries to do the same for classes as it does for variables and message selectors. Here I typed the character `w` after `Bot`, and Squeak, expecting a class name because of the uppercase `B` in `Botw`, indicates by turning the text red that there is no class in the system whose name begins `Botw`.

Summary

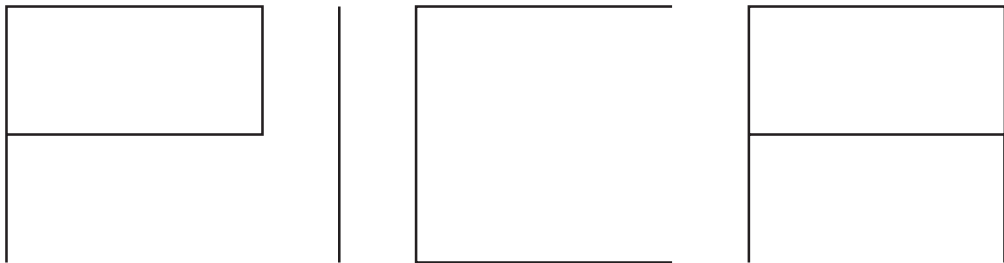
- To execute an expression. Press the **Do It All button** of the Workspace.
- A script is a sequence of expressions that performs a task.
- A message is composed of a message selector and possibly one or more arguments. Some message selectors do not take arguments, as in the message `send pica beInvisible`.
- Any message selector that ends with a colon requires additional information (one or more arguments), such as a length or an angle. For example, the message selector `turnLeft:` requires an argument whose value is a number representing the angle through which the robot should turn counterclockwise.
- To obtain a new object, you generally send the message `new` to a class. For example, `Bot new` creates a new robot. Other classes may understand different messages for producing new objects. For example, `Color yellow` asks the class `Color` to create a new yellow color object.
- A class is a factory for producing objects. Class names always start with an uppercase letter. For example, `Bot` is the factory for creating new robots, and `Color` is the color factory. The message `Bot new color: Color yellow` asks the `Bot` class to create a new robot, and then the color factory is asked to create a yellow color object. Finally, the message `color:` is sent to the new robot with the yellow color object as argument, resulting in the new robot having its color changed to yellow.
- Message sends should be separated by a period. A terminal period after the last message send is not required. Here is an example of four message sends separated by three periods:

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

- To send multiple messages to the same object use a semicolon to separate the messages, as in `aBot message1 ; message2`. For example, `pica go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90` send the sequence of four messages (1) `go: 100`, (2) `turnLeft: 90`, (3) `go: 200`, (4) `turnLeft: 90` to the robot named `pica`.



Of Robots and Men



In this chapter I describe the creation of robots and the different types of movements that robots know about and are capable of performing. I offer some simple experiments for you to perform, so that you can practice what you have learned in the previous chapters. I also will show you how robots can change direction along the fixed, or *absolute*, points of the compass.

Creating Robots

In the previous chapter you created *a* robot, not *the* robot. That is, robots are not unique, and you can create as many robots as you want. Script 3-1 creates two robots: pica and daly.

Script 3-1. *Two robots are born.*

```
| pica daly |
pica := Bot new.
daly := Bot new.
pica color: Color yellow.
daly jump: 100.
```

The second line creates a robot named pica as in Script 2-1. The third line creates a new robot that we refer to using the variable daly. (Just as pica's name is in homage to Pablo Picasso, that of daly is to honor Salvador Dali.) Both robots are created at the same location on the screen. In line four, we tell pica to change its color to yellow so that we can distinguish the two robots.

Smalltalk is an object-oriented programming language, as I have mentioned. This means not only that we can create objects and interact with them, but that objects can create other objects and communicate with them. Moreover, in Smalltalk, there are special objects, called *classes*, that are used to create objects. Sending the message `new` to a class creates an object described by its class. Sending the message `new` to the `Bot` class creates a robot.

To understand what classes are, imagine a class as a sort of factory. A factory for creating tin boxes might turn out large numbers of generic boxes, all of the same size, color, and shape. After they have been manufactured, some boxes might be filled with biscuits, while others might be crushed. When one box is crushed, other boxes are not affected. The same holds for objects created inside Squeak. In our case, daly did not change color, but pica did, while pica did not move, but daly did. You can think of a class as a factory able to produce unlimited supplies of objects of the same type. Once produced, each object exists independently of the others and can be modified as one wishes.

In Smalltalk, class names always begin with an uppercase letter. That's why the name of the robot class is `Bot` with an uppercase "B." Notice that in the command `Color yellow`, the word `Color` is written with an uppercase "C." That is because `Color` is a class, and what it manufactures is color objects. By specifying the color name, you get an object of the color you want. (The expression `Color yellow` is actually a short form for creating a yellow color object. First, a color object is created by sending the message `new` to the class `Color`, and then some extra messages define the color to be yellow.)

Important! A class is a factory that manufactures objects. Sending the message `new` to a class creates an object of that class. Class names always start with an uppercase letter. Here `Bot` is the name of the factory for creating new robots, and `Color` is the factory for colors.

Thus the command `Bot new color: Color blue` sends a message to the `Bot` class to create a new robot and then sends a message to the new robot to color itself with the color blue.

Drawing Line Segments

Asking a robot to draw a line is rather simple, as you already saw in the previous chapter. The message `go: 100` tells a robot to move ahead 100 pixels, and the robot leaves a trace during its move. However, when you draw, even if you are an expert Chinese or Japanese calligrapher, you need to lift the brush from time to time. For this purpose, a robot knows how to jump; that is, a robot can move without leaving a trace. A robot understands the message `jump:`, whose argument is the same as that for `go:`; namely, it is a distance, given in pixels. Script 3-2 draws two segments. To keep the picture uncluttered, I have kept the robots out of the illustration using the message `beInvisible`.

Script 3-2. *Pica is created and then draws two lines.*



```
| pica |
pica := Bot new.
pica go: 30.
pica jump: 30.
pica go: 30.
```

Experiment 3-1 (Creating and Moving a Robot)

Experiment by changing the values in the previous script.

Experiment 3-2 (SOS)

Write a script that draws the message “SOS” in Morse code. (In Morse code, an “S” is represented by three short lines, and an “O” is represented by three long lines, as shown in figure below.



Changing Directions

A robot can orient itself along the eight principal directions of the compass, as shown in Figure 3-1. The directions are like those on a standard map: east is to the right, west to the left, north up, and south down. These directions are *absolute*, which means that regardless of the direction in which a robot is currently pointing, if you tell it to point east, the robot will point to the screen's right, not to the robot's right. To point a robot in a given absolute direction, just send it a message with the name of the direction. Thus, to tell pica to face south, you simply type `pica south`.

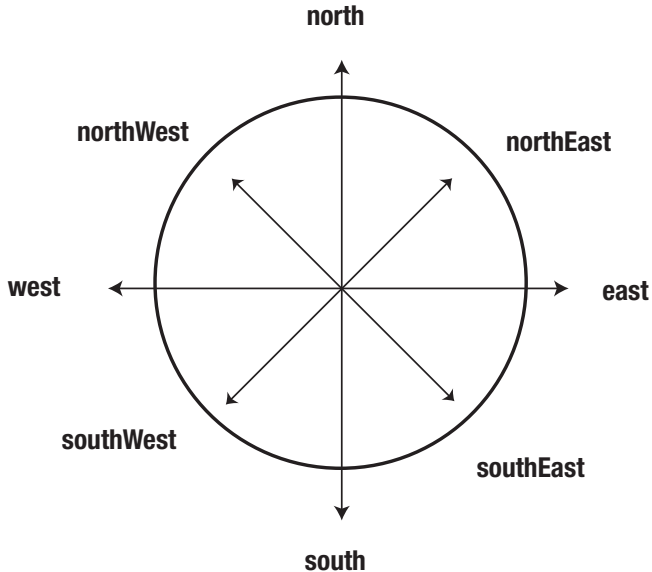


Figure 3-1. *The default absolute directions of the compass in which a robot can point.*

Robots understand the following compass direction messages: east, north, northEast, northWest, south, southEast, southWest, and west. In the next chapter, I will show you how to make a robot turn relative to its current position through an arbitrary angle.

Script 3-3 illustrates the four cardinal directions with four different robots; here Picasso and Dali are joined by Paul Klee and Alfred Sisley. Except for pica, who remains in the default direction east in which it was created, each robot is oriented in a different direction before being told to move.

Script 3-3. *A gaggle of robots go walking.*

```
| pica daly klee sisl |
pica := Bot new.
pica color: Color green.
pica go: 100.
daly := Bot new.
daly north.
daly color: Color yellow.
daly go: 100.
klee := Bot new.
klee west.
klee color: Color red.
klee go: 100.
sisl := Bot new.
sisl south.
sisl go: 100.
```

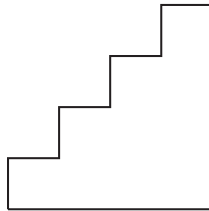
You can use these orientation methods to make more complex drawings.

Experiment 3-3 (A Square)

As a first exercise, draw a square with sides of length 50 pixels. Then draw another square of side length 250 pixels.

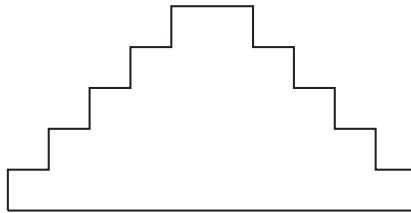
Experiment 3-4 (A Staircase)

You are not limited in your robot drawings to squares. You can create a wide range of geometrical figures. For example, here is a drawing of a small staircase. Write a script to reproduce this drawing.



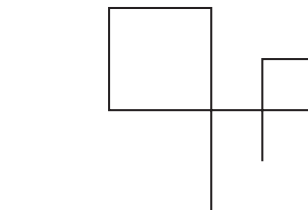
Experiment 3-5 (The Step Pyramid of Saqqara)

Now you are ready to spread your architectural wings and draw a schematic side view of the step pyramid of Saqqara, built around 2900 B.C.E. by the architect Imhotep. Write a script to draw a side view of this pyramid, as shown in the figure. The pyramid has four terraces, and its top is twice as large as each terrace.



Experiment 3-6 (Abstract Art)

Write a script to draw the picture shown in the figure below.



The ABC of Drawing

Even though you don't yet have much control over the direction of a robot's line segments, you can start programming pica to write letters. Script 3-4 draws a rather primitive letter "A."

Script 3-4. *The letter A is drawn.*



```
| pica |
pica := Bot new.
pica north.
pica go: 100.
pica east.
pica go: 100.
pica south.
pica go: 100.
pica north.
pica go: 50.
pica west.
pica go: 100
```

Drawing a letter "C" is no more difficult. You can even write a script to spell out "pica."

Experiment 3-7 (PICA)

Draw the name "PICA" as shown at the start of the chapter. To separate the individual letters, you should use the command `jump:`.

Remark One could argue that Script 3-4 could be improved. For example, the bottom half of the right-hand vertical line of the “A” is drawn twice, since the robot goes back over this segment—once going south, once going north—in order to get into position to draw the horizontal bar. Deciding on the best approach to solving a programming problem can be a difficult proposition. There are many issues to be considered, such as speed, complexity, and readability of the code, and these questions will have different answers depending on the programming language and the methods used. However, one approach you might consider is to start off by choosing the simplest solution. Then if you are dissatisfied because the program is too slow or doesn’t have the particular bells and whistles you want, you can always modify it to speed it up or add other enhancements.

Controlling Robot Visibility

You can control whether a robot is to be displayed using the messages `beInvisible` and `beVisible`. The message `beInvisible` hides the receiver of the message. A hidden robot acts exactly like a normal one; it just doesn’t show where it is. Be careful not to use the method `hide`, which is defined by Squeak for its own purposes and can damage the robot environment if used improperly. The message `beVisible` makes the robot receiving the message visible. A newly created robot is visible by default.

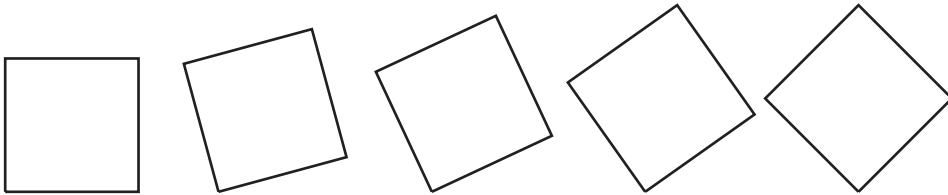
Summary

The following table summarizes the expressions and messages encountered in this chapter.

Expressions / Messages	Description	Example
<code>Bot new</code>	Create a robot.	<code>pica := Bot new</code>
<code> x y </code>	Declare variables to be used in the script.	<code> pica </code>
<code>jump: anInteger</code>	Tell a robot to move forward a given number of pixels without leaving a trace.	<code>pica jump: 10</code>
<code>go: anInteger</code>	Tell a robot to move forward a given number of pixels while leaving a trace.	<code>pica go: 10</code>
<code>beInvisible</code>	Tell a robot to be invisible.	<code>pica beInvisible</code>
<code>beVisible</code>	Tell a robot to be visible.	<code>pica beVisible</code>
<code>east, northEast, north, northWest, west, southWest, south, southEast.</code>	Tell a robot to point in the given direction.	<code>pica north</code>
<code>Color colorname</code>	Create the color <i>colorname</i> .	<code>Color blue</code>
<code>color: aColor</code>	Ask a robot to change its color.	<code>pica color: Color red</code>



Directions and Angles



By now, you should be getting tired of drawing figures only in *fixed* directions. In this chapter you will learn how to change the direction in which a robot points, allowing the robot to point in *any* direction, to turn through any angle relative to its current position, and therefore to draw lines in any direction. If you already understand clearly what an angle is and how to measure angles in degrees, you may skip the section “The Right Angle of Things” and then proceed to the examples and experiments in the section “Simple Drawings.”

I will begin by presenting the elementary messages for changing direction that robots understand. I am going to hide the robots from the illustrations using the message `beInvisible` so that you can get clearer pictures.

Right or Left?

In the previous chapter, you learned that a robot can be made to face in different directions using the messages `east`, `north`, `northEast`, `northWest`, `south`, `southEast`, `southWest`, and `west`. However, with these messages you cannot change the direction of your robot through an arbitrary angle, such as 15 degrees. In addition, you cannot turn a robot through, say, a quarter turn relative to its current direction.

To turn a robot through a given angle you should use the two methods `turnLeft:` and `turnRight:`, which tell a robot to turn to the left or the right. As the colon at the end of each method name indicates, these two methods expect an argument. This argument is the angle through which the robot should turn relative to its current position. That is, the argument is the difference between the robot's direction before the message is sent and its direction after the message is sent. This angle is given in degrees. For example the expression `pica turnLeft: 15` asks `pica` to turn to the left fifteen degrees from its current direction, and `pica turnRight: 30` turns `pica` to the right thirty degrees from its current direction. Figure 4-1 illustrates the effect of the messages `turnLeft:` and `turnRight:`, first when a robot is pointing to the east, and second when a robot is pointing in some other direction.

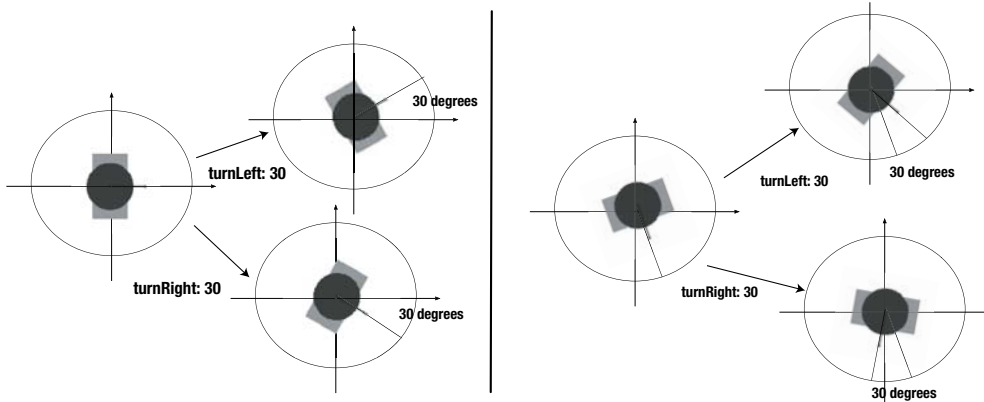


Figure 4-1. *Left: A robot facing east turns left or right through 30 degrees. Right: A robot facing in some other direction turns left or right through 30 degrees.*

As you practice turning robots through various angles, keep in mind that when a new robot is created, it always points to the east, that is, to the right of the screen.

Experiment 4-1 (Mystery Scripts)

Scripts 4-1 and 4-2 present problems in which you are to guess what the created robot will do. After studying these two scripts, experiment with them by changing the angle values, for example to determine what angle turns the robot through a quarter circle, a half circle, or a full circle. If you need to review the notion of angle, read the section “The Right Angle of Things” before continuing.

Script 4-1. *What does pica do? (Problem 1)*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 45.
pica go: 50.
pica turnLeft: 45.
pica go: 100
```

Script 4-2. *What does pica do? (Problem 2)*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnRight: 60.
pica go: 100.
pica turnLeft: 60.
pica go: 100
```

A Directional Convention

In mathematics, it is a general convention that rotation through a negative angle is construed as clockwise, while one with a positive angle is in the counterclockwise direction. You can also make use of this mathematical convention by using the message `turn:`. Hence, the message `turnLeft: aNumber` is equivalent to the message `turn: aNumber`, while the message `turnRight: aNumber` is equivalent to `turn: -aNumber`, where $-aNumber$ is the negative of $aNumber$. This relationship is depicted in Figure 4-2.

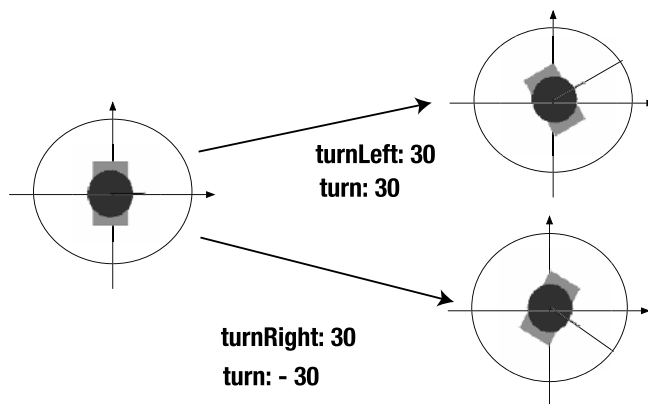


Figure 4-2. *Turning through a 30-degree angle starting from the direction east*

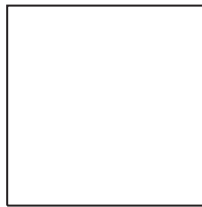
Absolute Versus Relative Orientation

You should now feel confident that you can ask a robot to execute any drawing consisting of straight lines. Before going further, be certain that you understand the difference between orienting a robot *absolutely* using the methods `north`, `south`, `southEast`, `east`, etc., and using the methods `turn:`, `turnLeft:`, and `turnRight:` to orient the robot *relative* to its current orientation.

Experiments 4-2, 4-3, and 4-4 will help you to solidify your understanding of this difference.

Experiment 4-2 (A Relative Square)

Write a script to draw a square using the method `turnLeft:` or `turnRight:`.



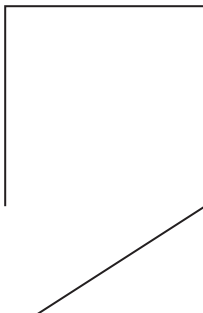
Experiment 4-3 (Tilting the Square)

Modify your script from Experiment 4-2 by adding the line `pica turnLeft: 33.` before the first line containing the message `go: 100.` You will obtain a square again, but it is tilted 33 degrees from the previous one.

Experiment 4-4 (A Broken Square)

Finally, execute Script 4-3, which attempts to draw a tilted square using the methods `north`, `south`, `east`, and `west` that we presented in the previous chapter.

Script 4-3. *A broken square*



```

| pica |
pica := Bot new.
pica turnLeft: 33.
pica go: 100.
pica north.
pica go: 100.
pica west.
pica go: 100.
pica south.
pica go: 100.

```

Do you still obtain a square? No! The first side drawn by the robot is slanted, whereas the other sides are either horizontal or vertical. The script that you wrote for Experiment 4-3 and Script 4-3 demonstrate the crucial difference between *relative* and *absolute* changes in direction:

- The methods `north`, `south`, `east`, and `west` change direction in an *absolute* manner. The direction in which the robot will point *does not depend* on the current direction in which it is pointing.
- The methods `turnLeft:` and `turnRight:` change direction in a *relative* manner. The direction in which the robot will point *depends* on its current direction.

Figure 4-3 shows the equivalence between relative moves starting with a robot pointing to the east and absolute moves. As you know, this equivalence is valid only if the robot is pointing east and not if it is pointing in any other direction. By the way, note that turning the robot 180 degrees points it in the opposite direction; this trick is often used in scripts.

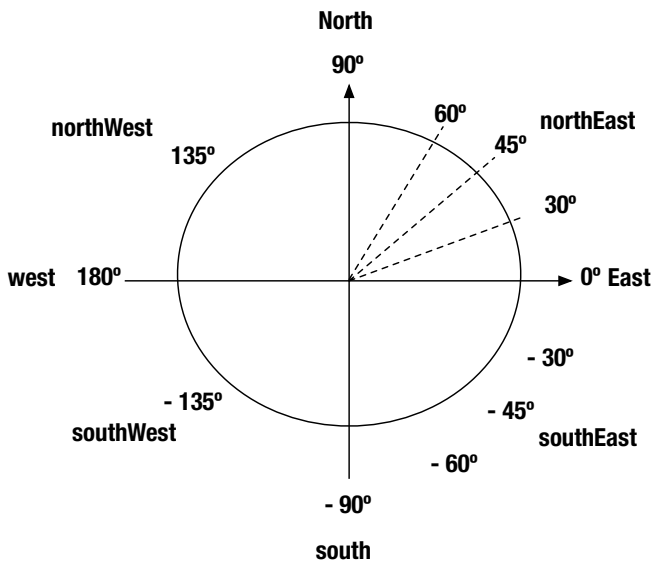
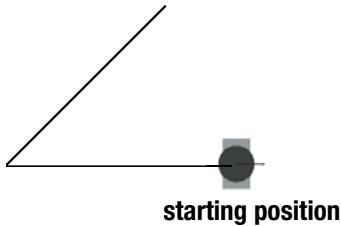


Figure 4-3. Comparing absolute and relative orientation starting from the easterly direction

The Right Angle of Things

As you know by now, a newly created robot is pointing east, that is, toward the right-hand side of the screen. If we ask this robot to turn left by 90 degrees, it will end up heading north. If instead, we ask it to turn right by 90 degrees, it will end up heading south. Script 4-4 illustrates the result of a turn left by 45 degrees. To help you in following the script, the accompanying figure shows the robot's starting position.

Script 4-4. *Moving through angles (1)*

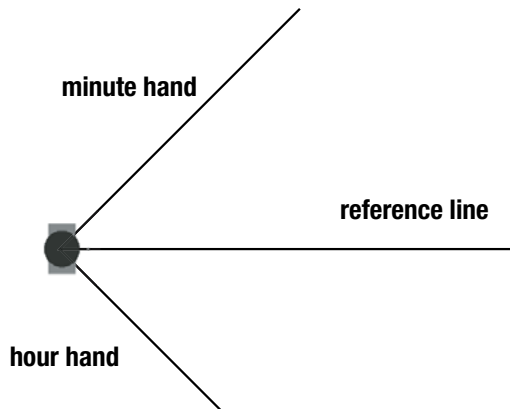


```
| pica |
pica := Bot new.
pica west.
pica go: 100.
pica east.
pica turnLeft: 45.
pica go: 100.
```

The first part of Script 4-4, up to the line `pica east`, draws a horizontal line, which will act as a reference line to indicate the easterly direction. The last part draws a line in the direction 45 degrees to the left of the easterly direction. You can vary the value of the angle to see what sort of angles other numbers of degrees represent. Try the values 60, 120, 180, 240, 360, and 420. In particular, note that a turn by 180 degrees amounts to turning the robot in the opposite direction from which it is pointing.

Do you see any difference between arguments of 60 and 420? They represent the same angle! Any two angle values whose difference is 360 or any multiple thereof are equivalent because 360 degrees represents a complete circle. Try an angle value of 1860 ($1860 = 60 + 360 \times 5$). The result is the same as you obtained with angle values 60 and 420. So keep in mind in dealing with angles that a robot's orientation does not change by adding one or more full turns to the orientation.

Now let us have some fun with the method `turnRight:`. Script 4-5 draws the hour and minute hands of a clock together with a reference line. It uses two robots, which you can use to investigate the correspondence between a left turn and a right turn. I have added comments surrounded by quotation marks and have employed a variety of font effects to help you to identify the different parts of the script. Note that you do not have to type these comments, since they are not executed.

Script 4-5. *Moving through angles (2)*

```

| pica daly |
pica := Bot new.
pica jump: 200.           "drawing the reference line"
pica turnLeft: 180.
pica go: 200.
pica turnLeft: 180.
pica color: Color blue.
pica turnLeft: 45.       "drawing the minute hand"
pica go: 150.
daly := Bot new.
daly color: Color red.
daly turnRight: 45.    "drawing the hour hand"
daly go: 100.

```

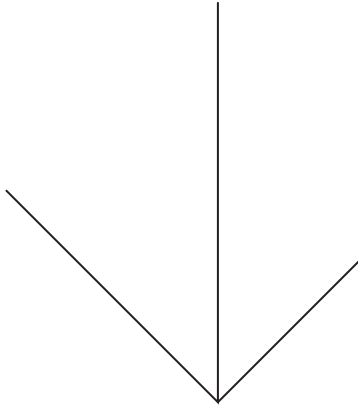
In Script 4-5, the code in italics draws the reference line—that is, the line representing the direction of the robot before a turn method is executed—using the fact that a turn through 180 degrees amounts to turning around to point in the opposite direction. The reference line is also the longest line drawn. Thus, the reference line will still be visible if the lines drawn by the robots fall on top of it. The text in normal roman font following the italics is the code that draws the minute hand (using *pica*) and in bold, the code drawing the hour hand using the robot *daly*.

Experiment 4-5 (Moving Clock Hands)

Experiment with different angle values for each of the two robots; that is, change the angle values for the two turn methods. Then, compare the effect of the method `turnLeft: 60` (for *pica*) and `turnRight: 300` (for *daly*). You can see that turning left 60 degrees yields the same result as turning right 300 degrees. This is so because the sum of the two values is 360 degrees, that is, a full circle.

Now let us see what happens when the robot turns from another direction. Here is the same script as Script 4-4 but showing the effect of turning from the north. In this script we are replacing daly by another robot, *berthe*, who honors the French impressionist painter Berthe Morisot.

Script 4-6. *Moving through angles (3)*



```
| pica berthe |
pica := Bot new.
pica north.
pica jump: 200.
pica turnLeft: 180.
pica go: 200.
pica turnLeft: 180.
pica color: Color blue.
pica turnLeft: 45.
pica go: 150.
berthe := Bot new.
berthe north.
berthe color: Color red.
berthe turnRight: 45.
berthe go: 100.
```

Experiment 4-6 (Changing the Reference Direction)

Continue to experiment with Script 4-6 by changing the reference direction. For the comparison to be meaningful, you have also to orient *berthe* in the same direction as *pica* after creating her. Try any angle values you like and try to predict what the resulting drawing will look like before executing the script. Continue experimenting with the script until your predictions are accurate.

Note that you should always be able to predict what is going to happen before executing a script, because a computer blindly executes all valid statements, even the silliest ones.

A Robot Clock

I have mentioned that the lines drawn in Script 4-6 are akin to the hands of a clock. The analogy between time and angles is a good one, for the notion of degrees is strongly correlated with that of hours. Ancient civilizations discovered the notion of time by measuring the angle of the sun (or a star) relative to a reference direction. However, a script like Script 4-6 allows you to place the hands in a position that does not indicate a real time of day. For example, you could draw a clock with the hour hand pointing north and the minute hand pointing south. But on a real clock, when the minute hand is pointing south, it is half past the hour, and so the hour hand should be halfway between two numbers on the clock's face.

Now you will study the relationship between the hour hand and the minute hand on a *real* clock that represents a *real* time of day.

Experiment 4-7 (A “Real” Clock)

Modify Script 4-6 as follows:

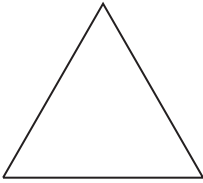
- Keep the direction of reference to the north (this is how Script 4-6 is written). This reference line indicates 12:00 noon or midnight.
- Use the method `turnRight`: for both robots. After all, the hands of a clock move clockwise, which is to the right.
- You can ask `pica` to draw the minute hand by multiplying the number of minutes after the hour that you wish to indicate by 6 (since during the 60 minutes in an hour, the minute hand travels the $6 \times 60 = 360$ degrees in a full circle). For example, to represent the minute hand for 20 minutes after the hour, you should use the expression `turnRight: 120` (since $120 = 6 \times 20$).
- You can ask `berthe` to draw the hour hand by multiplying the number of the hours you want to indicate by 30 (12 hours times 30 degrees per hour equals 360 degrees) and then adding one-half (0.5) of a degree for each minute after the hour, since in 60 minutes, the hour hand moves 30 degrees. For example, the hour hand is positioned for 2 o'clock with the message `turnRight: 60` ($60 = 30 \times 2$), while the time 4:26 requires the hour hand to be positioned with the message `turnRight: 133` ($133 = 30 \times 4 + 26 \times 0.5$).

Try to indicate a few times of your choice with this modified script.

Simple Drawings

To begin with, here is a script for drawing a triangle with three equal sides:

Script 4-7. *An equilateral triangle*



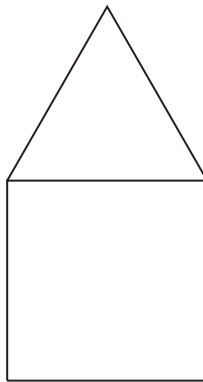
```
| pica |  
pica := Bot new.  
pica go: 100.  
pica turnLeft: 120.  
pica go: 100.  
pica turnLeft: 120.  
pica go: 100.  
pica turnLeft: 120.
```

The last line of code is not necessary for drawing the triangle; it serves to point *pica* back in his initial position.

Now, you are ready to draw a house.

Experiment 4-8

Draw a house as shown in the figure. Try to draw houses of different shapes.



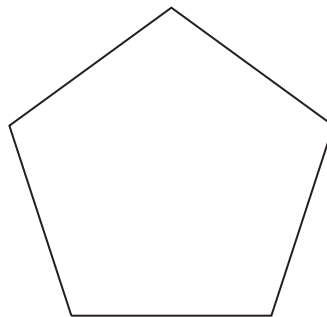
Regular Polygons

A regular polygon is a figure composed of line segments all of the same length and all of whose angles are equal. An equilateral triangle is a regular polygon with three sides. A square is a regular polygon with four sides. For example, Script 4-7 draws an equilateral triangle whose side length is 100 pixels. It is obtained by telling `pica` to go forward 100 pixels and then turn 120 degrees left, and then repeating these two messages two more times so that they are executed three times altogether.

You can program a robot to draw a regular polygon with any number of sides by asking it to move a certain length and then turn left or right by 360 degrees divided by the number of sides; this sequence must be repeated as many times as there are sides. Note that the last turn by the robot can be omitted, since the robot has drawn the last line of the polygon.

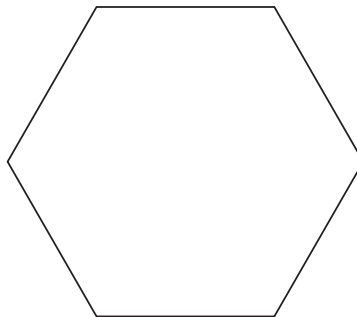
Experiment 4-9

Draw a regular pentagon (a regular polygon with five sides), as shown in the figure, with sides of length 100 pixels.



Experiment 4-10

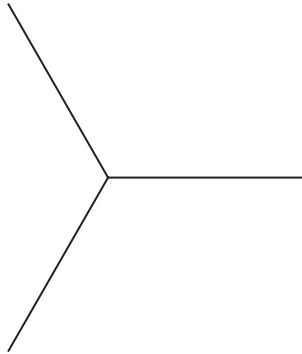
Draw a regular hexagon (a regular polygon with six sides), as shown in the figure, with sides of length 100 pixels.



If you are just curious to see how far you can go with this process, you can use the cut and paste feature of the Bot workspace to generate a regular polygon with a large number of sides. If you are in the mood, go on increasing the number of sides. However, in Chapter 7, I will show you how you can type a sequence of expressions once and then have them repeated over and over.

Experiment 4-11

Draw the three-spoked figure shown below.



Summary

- A robot can be oriented *relative* to its *current* direction using the methods `turnLeft:` and `turnRight:`.
- The parameter given to the methods `turnLeft:` and `turnRight:` is given in degrees.
- Turning 360 degrees corresponds to a turn through a full circle.
- Turning 180 degrees corresponds to a turn through a half circle.
- Angle values whose difference is a multiple of 360 degrees are equivalent.

Here is a list of the methods that you have learned about in this chapter.

Method	Syntax	Description	Example
turnLeft:	turnLeft: aNumber	Tell the robot to change its direction by a given number of degrees to the left.	pica turnLeft: 30
turnRight:	turnRight: aNumber	Tell the robot to change its direction by a given number of degrees to the right.	pica turnRight: 30
turn:	turn: aNumber	Tell the robot to change its direction to a given number of degrees following the mathematical convention that a turn is to the left if the number is positive and to the right if it is negative.	pica turn: 30
beInvisible	beInvisible	Hide the receiver.	pica beInvisible
beVisible	beVisible	Show the receiver.	pica beVisible



Pica's Environment

In this chapter, I will present pica's environment and show you how to obtain tools and save your scripts. I will also return to the notion of messages and show that you can ask the environment not only to execute a message, but also to print the *result* of the message execution.

The Main Menu

When you click on the background you get the main menu of the environment, as shown in Figure 5-1.

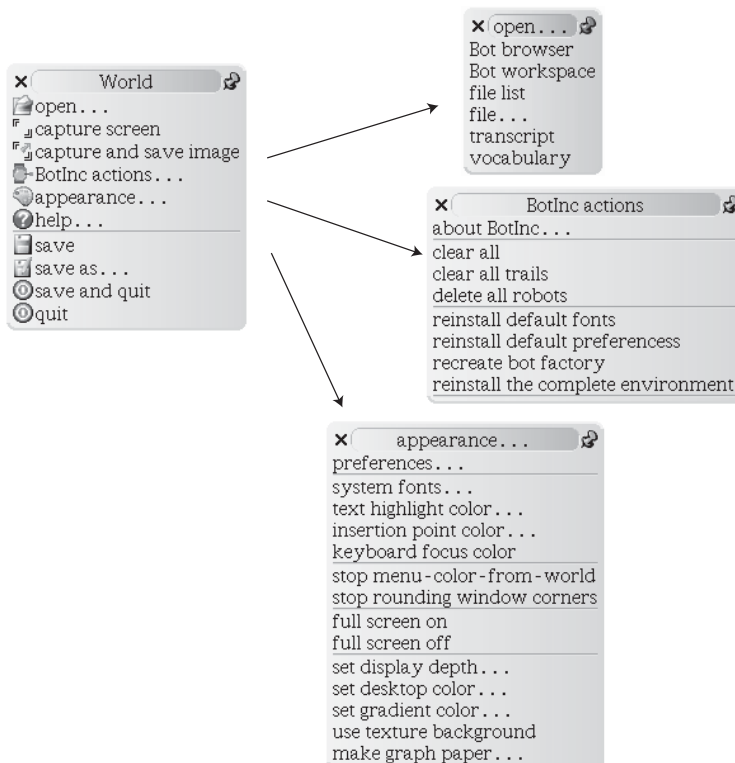


Figure 5-1. Menu options of the environment

If you want to know what a particular menu item does, simply move the mouse pointer over it for a second, and voilà! a balloon should pop up describing the item. The main menu gives access to five main groups of functionalities: access to tools, screen capture, access to some robot behavior, appearance, and saving the environment. The submenus are grouped as follows:

- The **open...** menu collects several tools such as the robot code browser, the Bot workspace, a file browser, and other tools that I will present as needed.
- The **BotsInc actions** menu collects several actions such as indicating the version of the environment and clearing all robots and their traces, as well as some actions to reinstall the environment if needed: reinstalling default preferences resets the preferences that you may have modified using the appearance menu to their default values.
- The **appearance...** menu collects actions that change the appearance of the environment such as fonts used, full-screen mode, and background color.

Obtaining a Bot Workspace

If you happen to close the default Bot workspace, don't worry. You can get a new one easily from the dark blue flap, as shown (though not in blue) on the left side of Figure 5-2, or from the main menu, as shown in Figure 5-1. To install a new Bot workspace in the working flap, open the working flap (bottom flap) and drop the Bot workspace from the blue flap into the bottom flap.

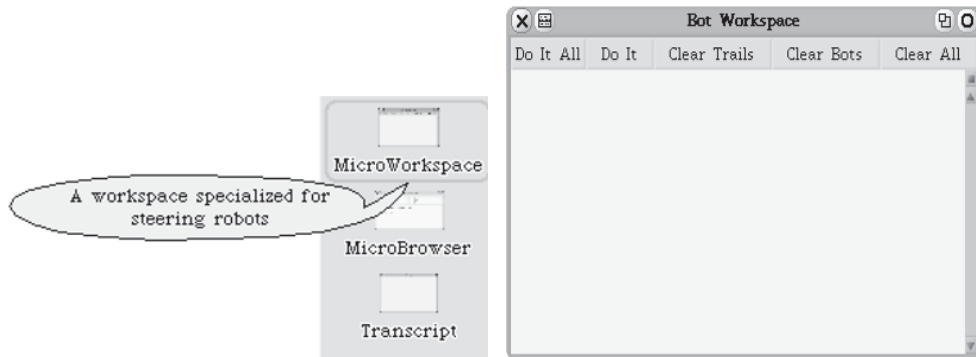


Figure 5-2. Obtaining a new Bot workspace from the flaps

The dark blue flap contains other tools that we are going to use in the future. The second tool is basically a code browser that you will use when you define new robot methods.

The environment contains a simple tool (Figure 5-3) that lists the most important messages that a robot can understand. You can obtain access to this tool via the **open...vocabulary** menu or the **help** menu (**open vocabulary**). The vocabulary pane lists the messages, grouped according to type. For example, the messages east, north, and so on are listed under **absolute directions**.

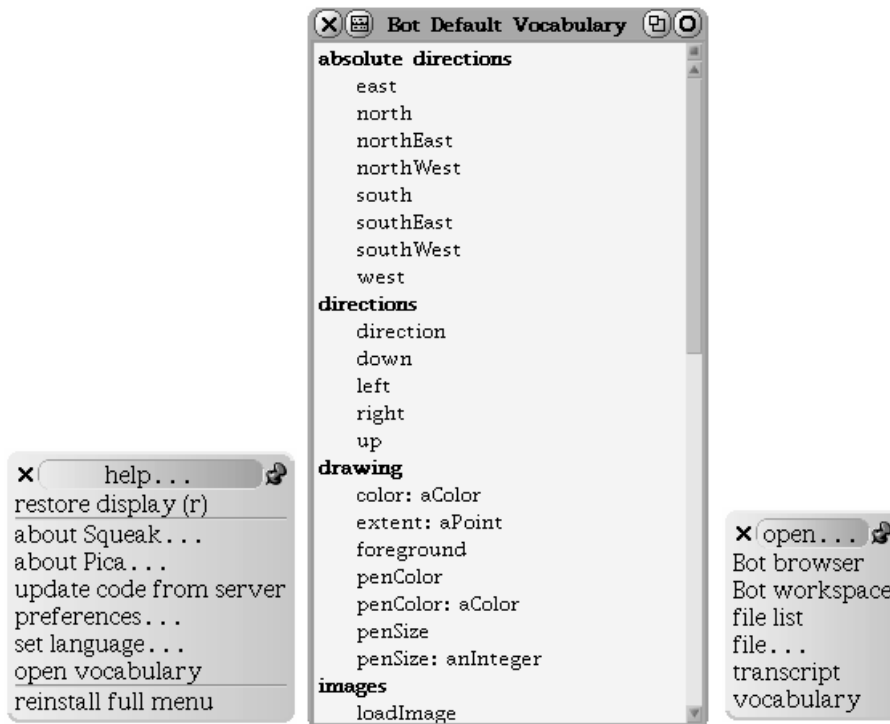


Figure 5-3. The most important messages for robots

Interacting with Squeak

Interaction with Squeak is based on the assumption that you have a three-button mouse, though there are button equivalents for a Windows two-button mouse or Macintosh one-button mouse, as shown in Table 5-1. Each button is associated with a logical set of operations. The left button is for obtaining contextual menus and for pointing and selecting, the middle button is for window manipulation (bringing a window to the front or moving it), and the right button is for obtaining handles, which are small colored and round buttons floating around graphical elements (as shown in Figure 5-4). Collectively, the handles are called a *halo*. The handles are useful, for they allow you to interact directly with the robot. I will present them in detail in the next chapter.

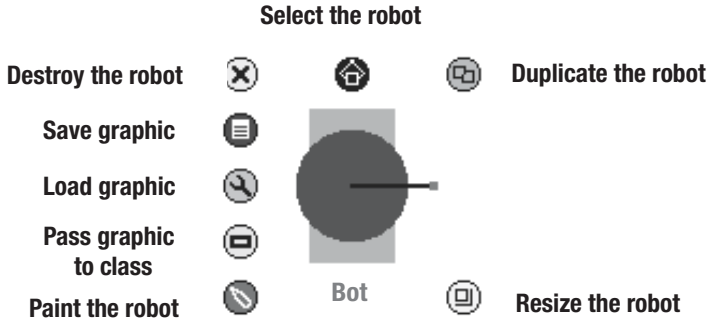


Figure 5-4. Right clicking on a robot brings up its halo of handles.

Table 5-1. Mouse Button and Key Combinations

	Pointing and Selecting	Context-Sensitive Menu	Open the Halo
Three buttons:	Left click	Center click	Right click
Windows 2-button equivalent:	Left click	Alt-left click	Right click
Mac 1-Button equivalent:	Click	Option-click	Command-click

Using the Bot Workspace to Save a Script

The Bot workspace has five buttons and a menu that allow you to save scripts. The button **Do It All** executes the entire script contained in the workspace. The button **Do It** executes the part of the script in the workspace that is currently selected. The button **Clear Trails** clears only the robot trails without removing the robots themselves. The button **Clear Robots** removes only the robots without clearing their trails. The button **Clear All** removes all the robots and their trails.

Once you have written a script, you may wish to save it to a file for future use. The Bot workspace provides a way of saving and loading files via the workspace menu. Click on the contents of the workspace to bring up its associated menu, as shown in Figure 5-5. The menu item **save contents** will save the complete contents of the workspace into a file. Selecting this menu item brings up a dialog box, as shown in the figure. Note that the system checks whether a file with the same name already exists. If such a file already exists, the system gives you the choice of overwriting the file or saving it under another name.

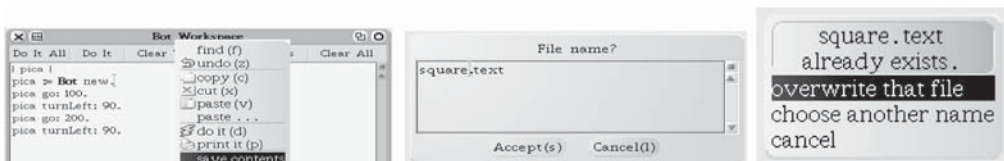


Figure 5-5. Left: Bot workspace menu options. Middle: Specifying the name of the file in which the script is to be saved. Right: If a file already exists, you can overwrite it or rename it.

Loading a Script

To load a script, you have to use a file list, a tool that allows you to select and load different files into Squeak. You can obtain a file list by selecting the menu item **open... file list** from the main menu. A file list comprises several panes. The top left pane allows you to navigate through volumes and folders; each time you select an item in this pane, the top right pane is updated. It shows all the files contained in the folder that you selected in the left pane. When you select a file in the right pane, the bottom pane automatically displays its contents. Figure 5-6 shows that we are in the folder `Bot` testing, in which the file `square.text` is selected.

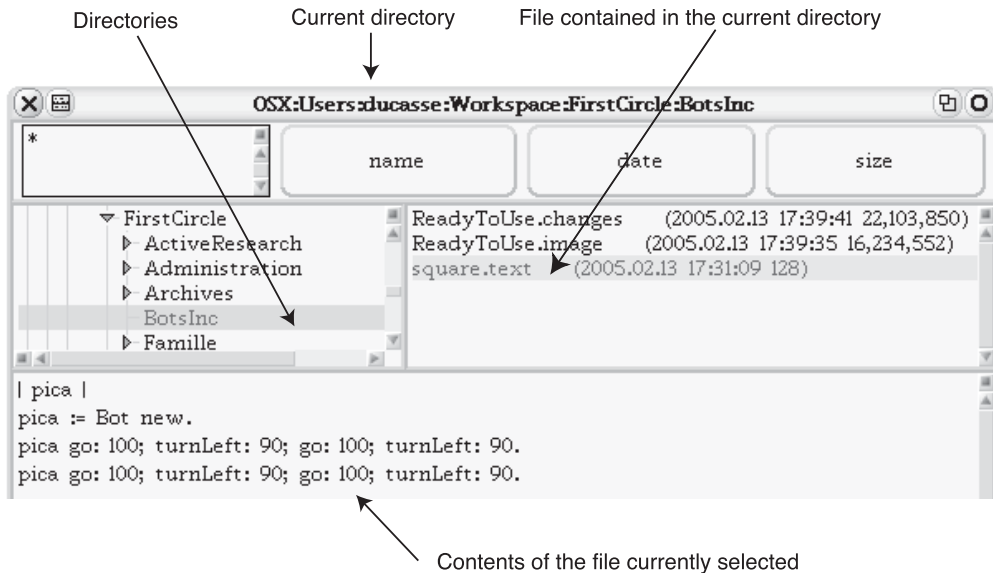


Figure 5-6. The file list is open to the script `square.text`

To load a script, you simply have to copy the contents of the bottom pane using the menu item **copy** and paste it into the Bot workspace using **paste**, just as you would in any text editor.

Capturing a Drawing

To keep a record of your drawings, you can use the screen capture feature of your computer. However, with some computers, screen capture is problematic. To avoid such problems, the environment offers a simple screen capture mechanism that works on any computer. Bring up the main menu by clicking on the background of the environment. The menu offers two items for capturing, named **capture screen** and **capture and save image**, as shown in Figure 5-7.

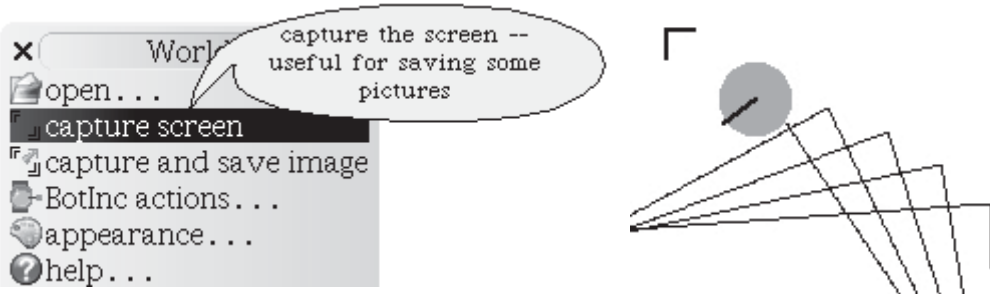


Figure 5-7. Left: Two possibilities for capturing and saving the capture. Right: The cursor has changed, indicating that Squeak is ready for the capture. Now click to position one corner of the rectangular region you want to capture.

The easier of the two options is to use the **capture and save image** menu item. When you select this item, Squeak shows that it is ready to capture by changing the cursor's shape to that of a corner, as shown on the right-hand side of Figure 5-7. Place the cursor at the corner of the rectangular region you want to capture, click, and drag the mouse to delimit the region you want. The region is displayed in the bottom left corner of the Squeak window, and Squeak prompts you for the name of the file without extension that it will save.

If you want to capture a region of the screen, use the menu item **capture screen**. In this case, Squeak will not prompt you to save the file, but instead, it creates a picture on the Squeak desktop, which you can save by first calling up the handles by right clicking on the screenshot. A number of different handles should appear around the image, as shown in Figure 5-8. Once the halo, that is, the group of handles, has appeared around your image, click on the red handle, which opens a menu of actions that you can apply to the image. Select **export...** and the format in which the image is to be saved. Squeak will prompt you for the name of the file. Note that you can import these files into Squeak by dropping them from the desktop onto the Squeak desktop.

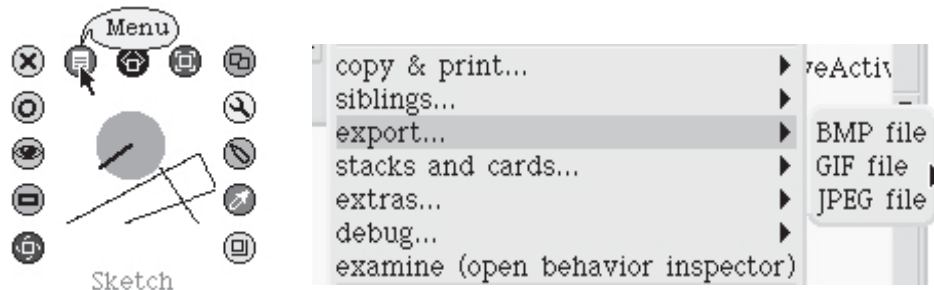


Figure 5-8. Call up the halo and choose the red handle menu item **export...** to save the image to disk.

Message Result

In Smalltalk, objects communicate only by sending and receiving messages to and from other objects. Once an object receives a message, it executes it, and additionally, it returns a result. A result is an object that the receiving object has returned to the sender. Communication between objects by means of messages is similar to communication between people by sending letters: Some letters that we receive require us to perform certain actions (such as a warning from the dogcatcher to keep our dog on a leash), while others might require us to sign an acknowledgment that we have received the letter (a certified letter).

In Squeak, the receiver of a message always returns a result, which by default is the receiver of the message. However, this result is often not of interest. For example, sending the message `go: 100` to a robot tells the robot to move 100 pixels in its current direction. But we have no use for the result returned, which in this case is the robot itself, so in this case, we ignore the result. In many cases, though, the result of a message execution is important. For example, the expression `2 + 3` sends the message `+ 3` to the object `2`, which returns the object `5`. Sending the message `color` to a robot returns its current color. The result of a message can be used as part of another message in a compound message. For example, when the expression `(2 + 3) * 10` is executed, the expression `(2 + 3)` is executed, whereby the message `+ 3` is sent to the object `2`, and this returns `5`. The result `5` is then used as the object to which a second message, `* 10`, is sent. Thus `5` is the receiver of the message, and it then returns the result `50`.

The Squeak environment allows you to execute messages without dealing with the message's result, and it also allows you to execute messages and print the returned message value. The following section will illustrate this difference in detail.

Note A *result* is an object that the receiving object returns to the object that sent a message. For example, `2 + 5` returns `7` and `pica color` returns `pica's color`, a color object.

In Figure 5-9, the expression `50 + 90` is selected, then using the menu the expression is executed, and the result, `140`, is printed on the screen.

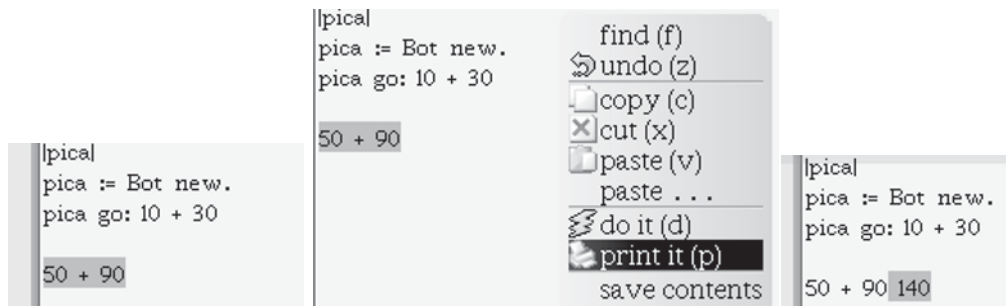


Figure 5-9. Left: Selecting the expression `50 + 90`. Middle: Opening the menu. Right: Executing the message and having the result printed.

Executing a Script

There are three ways of executing a script.

1. Using the buttons of the Bot workspace editor. In Chapter 2 you saw a simple way to execute your first script by pressing the **Do It All** button of the Bot workspace. But to execute a script, you can also *select* the text you want to execute with the mouse (the selection turns green) and then press the **Do It** button of the Bot workspace.
2. Using the menu. Select the part of your script you want to execute, as shown, for example, in Figure 5-10. Then open the menu by pressing the middle button of your mouse (or press the option key while clicking with the left button), and then choose the **do it (d)** or the **print it (p)** menu item as shown in Figure 5-9.

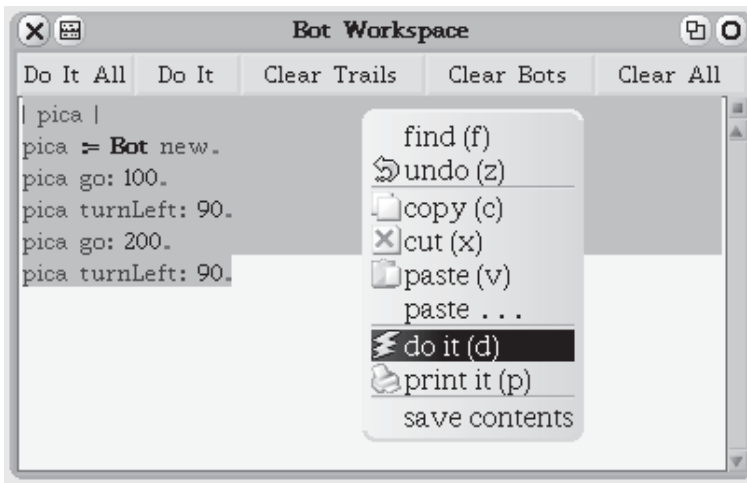


Figure 5-10. *Selecting a piece of a script and executing it explicitly using the menu*

3. Using keyboard shortcuts. Select a piece of text, then press command+D on a Mac or alt+D on a PC.

Hints

To automatically select all the text of a script, you can simply click at the start of the text (before the first character), at the end of the text, or on the line after the last expression. If you want to select a word, you can double click anywhere on the word. If you want to select a line, just double click at the beginning (before the first character) or end (after the last character) of the line.

Two Examples

When you execute the expression `pica color`, which asks the robot its color using the **do it (d)** menu item, the message `color` is sent and executed. However, you have the impression that nothing happens. This is because you have not asked the system to do anything with the result

of the message execution. If you are interested in the result of a message, you should use the menu item **print it (p)**, as shown in Figure 5-11. This has the effect of both executing the piece of code selected *and* printing the result returned by the last message in the code. In the figure, the expression `Bot new` is executed, and then the message `color` is sent to the newly created robot. The message `color` is executed, and the color of the receiving robot is returned and printed, as shown in Figure 5-12. The text `(TranslucentColor r: 0.0 g: 0.0 b: 1.0 alpha: 0.847)` tells us that the color of the robot is a translucent color composed of the three color components red, green, and blue.

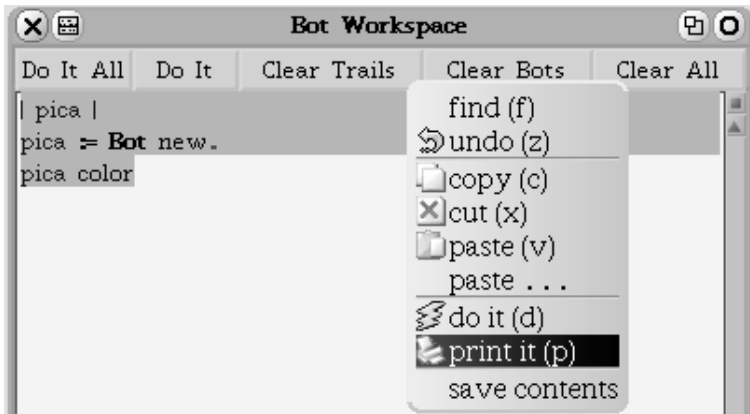


Figure 5-11. Open the menu and select the item **Print it (d)** to execute the selected piece of code and print the returned result.

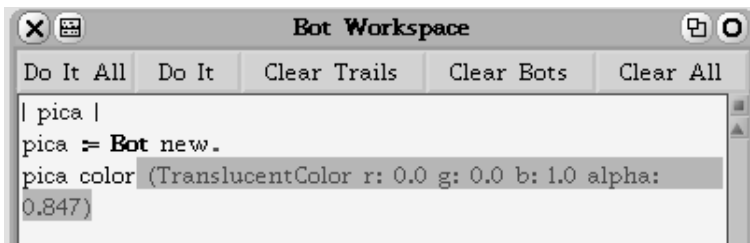


Figure 5-12. The result of the message is printed as a textual representation of a color.

Let's look at a final example to make sure that you understand when to use **print it**. When you execute the expression `100 + 20` using the menu item **do it (d)**, the message `+ 20` is sent to the object `100`, which adds 20. However, you do not see anything. This is normal, because in such a case the execution of the message `+ 20` returns a new number representing the sum, but you did not ask Squeak to print it. To see the result, you have to print the result of the message execution using the menu item **print it**. From now on, we will write “—Printing the returned value:” to indicate that we are using the print command to execute an expression and print its result, as shown in Script 5-1. Note that we will use this convention only when the result is important.

Script 5-1. *Printing the result of executing an expression*

```
(100 + 20) * 10  
-Printing the returned value: 1200
```

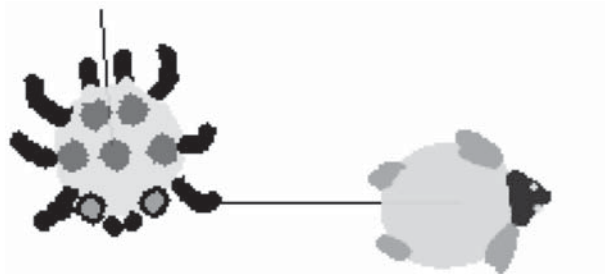
■ **Important!** There are two ways of executing an expression: (1) using the **Do It** menu item to execute an expression, and (2) using the **Print it** menu item to execute it and print the returned result.

Summary

- To execute an expression, select a piece of text representing one or several expressions and press the **Do It** button or select the menu item **do it** from the execution menu.
- A result is an object that you obtain from a message. For example, `pica color` returns the color of the robot.
- There are two ways of executing an expression, (1) using the **do it** menu item to execute an expression, and (2) using the **print it** menu item to execute it and print the returned result.



Fun with Robots



The basic look of a robot is rather simple. Wouldn't it be nice to be able to create robots that had a bit more pizzazz to them? Fortunately, you can create customized robots, and in this chapter, I will show you how you can change the shape, the pen size, and the color of your robots. You can make your robot look like an animal, a monster, or even the famous robot R2D2 from the movie *Star Wars*.

Robot Handles

You have learned about opening a message balloon for sending a message to a robot by clicking on that robot. Now you are going to learn about obtaining access to other robot functions such as duplicating, moving, and changing the look of a robot. These extra functionalities are available via the halo of handles, which, as was mentioned briefly in Chapter 5, you can beam up by right clicking (command clicking on a Mac) on a robot. The handles are the small, round icons that surround the robot like a halo, as shown in Figure 6-1. I will explain the functions of the different handles as they are needed. You can get information about a handle by letting your mouse rest over a handle; then a balloon pops up and explains the handle's purpose. For now, try to make a copy of the robot by clicking on the green (“duplicate the robot”) handle, move the robot by clicking on the black (“select the robot”) handle and dragging the robot, or destroy the robot using the pale pink (“destroy the robot”) handle with the “X.”

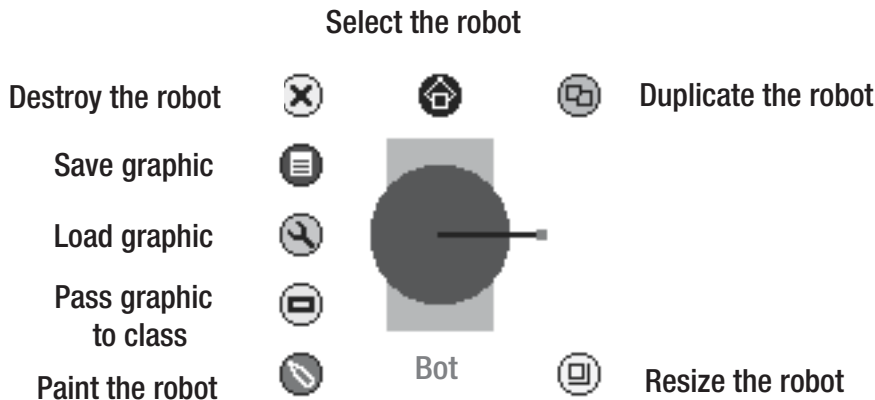


Figure 6-1. Right click (command click) on the robot to bring up the halo of handles.

Pen Size and Color

When our robots moved around the screen in previous chapters, they left a black trace in their wakes. But you are not limited to the default color black. You can change the color of a robot's pen by sending a robot the message `penColor:` with a color object as argument. One of the ways of obtaining a color object is to send a message with the name of a color to the class `Color`, which is a factory that makes color objects. For example, `Color blue` yields a blue color object, and `Color yellow` yields a yellow color object. Thus you can change the pen color of the robot `pica` to the color blue with the message `send pica penColor: Color blue`. I will explain more about colors in the following section.

You can also change the thickness of the robot's pen by sending the message `penSize:` with a number as argument. For example, `pica penSize: 5` orders `pica` to change his pen size to be 5 pixels wide. Script 6-1 draws a thick blue line of width 5 pixels.

Script 6-1. *Pica can draw a thick blue line.*

```
| pica |
pica := Bot new.
```

```
pica penColor: Color blue.
pica go: 100.
pica penSize: 5.
pica go: 100
```

Script 6-2 draws a sailor's spyglass by repeatedly increasing the pen size.

Script 6-2. *Pica draws a spyglass.*



```
| pica |
pica := Bot new.
pica go: 40.
pica penSize: 2.
pica go: 40.
pica penSize: 4.
pica go: 40.
pica penSize: 6.
pica go: 40.
```

You can change the color of the robot itself using the method `color:.` For example, the message `send berthe color: Color yellow` changes the robot `berthe`'s color to yellow. Script 6-3 tells `berthe` to change her color to yellow and then go forward 100 pixels, while `pica` is left behind with his default color and without moving.

Script 6-3. *Berthe changes her color and goes for a walk, while pica is left behind.*

```
| pica berthe |
pica := Bot new.
berthe:= Bot new.
berthe color: Color yellow.
berthe go: 100.
```

More about Colors

As previously mentioned, Squeak is an environment that is built from objects and that uses objects. Therefore, programming in Squeak amounts to creating objects and sending them messages. In particular, a *color* is an object created by the class `Color`. To obtain a color object, you send a message to the class `Color`.

Some color messages are named for the color they represent. For example, `Color red` causes the class `Color` to create a red color object. Here is the list of the predefined message selectors that you can send to the class `Color` to create that color: `black`, `veryVeryDarkGray`, `veryDarkGray`, `darkGray`, `gray`, `lightGray`, `veryLightGray`, `veryVeryLightGray`, `white`, `red`, `yellow`, `green`, `cyan`, `blue`, `magenta`, `brown`, `orange`, `lightRed`, `lightYellow`, `lightGreen`, `lightCyan`, `lightBlue`, `lightMagenta`, `lightBrown`, `lightOrange`, `paleBuff`, `paleBlue`, `paleYellow`, `paleGreen`, `paleRed`, `veryPaleRed`, `paleTan`, `paleMagenta`, `paleOrange`, and `palePeach`.

The `Color` class is like a real-life paint factory. Not only can it make a large number of standard colors, it can also create a customized color for you by combining different amounts of red, green and blue. Table 6-1 shows a few examples of how to create colors this way using the message `r: redAmount g: greenAmount b: blueAmount`. The arguments taken by the message selector `r:g:b:` should be decimal numbers between 0 and 1 representing the amounts of red, green, and blue to be combined. For example, the expression `Color r: 1 g: 0 b: 0` creates the same pure red color that you get from `Color red`. Using the same amount of each of the three colors produces a shade of gray. All ones produces white, and all zeros produces black.

Table 6-1. *Creating Colors with `Color r:g:b:`*

Color	r: (Red)	g: (Green)	b: (Blue)
red	1	0	0
light gray	0.1	0.1	0.1
yellow	1	1	0
white	1	1	1
black	0	0	0
gray	0.5	0.5	0.5
pale green	0.8739	1.0	0.8348

Finally, the method `fromUser` lets you pick a color from a palette on the screen, and then shows you that color's ingredients, as illustrated in Figure 6-2 (though you will have to imagine the colors). For that, you need to execute the expression `Color fromUser` using the **print it** menu to get the result of the selection printed.

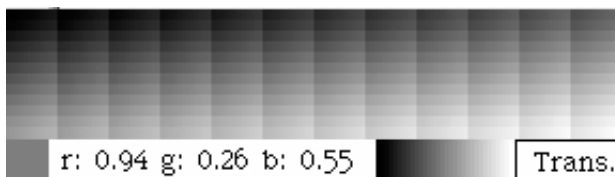


Figure 6-2. *Choose your color from a color palette with the message `send Color fromUser`.*

Changing a Robot's Shape and Size

You can change a robot's shape as well as its color. In addition to the default robot shape, two shapes, a circle and a triangle, are built into the `Bot` factory (but you can also draw the robot shape with a drawing tool, as shown in the next section). The message `lookLikeTriangle` gives a triangular shape to a robot. The message `lookLikeCircle` gives a circular shape to a robot. The default shape is produced by sending the message `lookLikeBot`.

Another aspect you can change is the size of a robot using the message extent: `widthAndHeight`, where the values of `widthAndHeight` represent the width and height of the rectangle in which the robot is drawn. The argument `widthAndHeight` is a pair of numbers, also called a *point* in Squeak. It is composed of two numbers separated by the @ symbol. For example, the point `50@100` represents a rectangle 50 pixels wide and 100 pixels tall.

Thus to create a robot named `bigpica` in the shape of a triangle that fits inside a square with dimensions `150@150`, you would first send `bigpica` the message `lookLikeTriangle` and then the message extent: `150@150`.

Figure 6-3 shows some robot shapes created using the built-in triangle and circle shapes, and Script 6-4 shows how to create robots of these sizes and shapes and move them into position as shown in the figure.

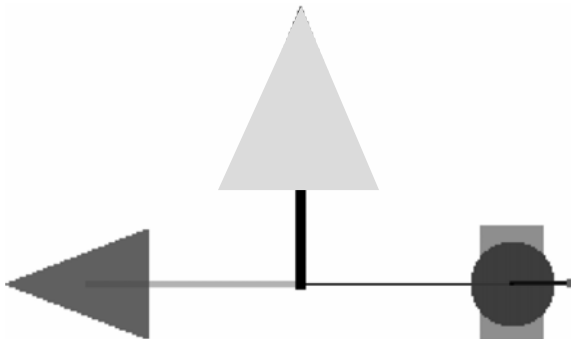


Figure 6-3. Robots can come in different shapes and sizes.

Script 6-4. Creating robots of different sizes and shapes (circles and triangles)

```
| pica daly bigpica |
pica := Bot new.
pica lookLikeTriangle.
pica west.
pica color: Color red.
pica penColor: Color green.
pica penSize: 3.
pica go: 100.
daly := Bot new.
daly extent: 60@60.
daly east.
daly go: 100.
bigpica := Bot new.
bigpica lookLikeTriangle.
bigpica extent: 150@150.
bigpica penSize: 5.
bigpica north.
bigpica go: 80.
```

Drawing Your Own Robot

Squeak lets you draw a customized robot. You can even create a robot that looks like one of the figures shown at the beginning of this chapter. I will now describe step by step how to draw your own robot.

Step 1: Open the painting tool via the red handle. The first step is to open the painting tool that is included in Squeak. Right click (or command click for Mac) to beam up the halo around the robot that you want to paint, as shown in Figure 6-4. Click on the red handle, the one with the icon of a pen inside. This will open the painting editor, which is depicted in Figure 6-5. Do not worry about the other handles. Note that if you have already drawn a graphic, that graphic will be shown inside the painting tool.

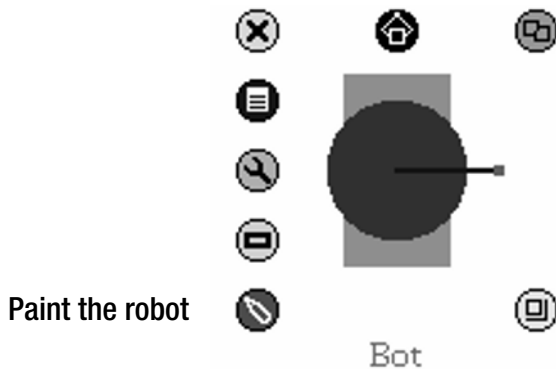


Figure 6-4. Right click (or command click) to obtain the halo. Choose the painting editor from the red handle.

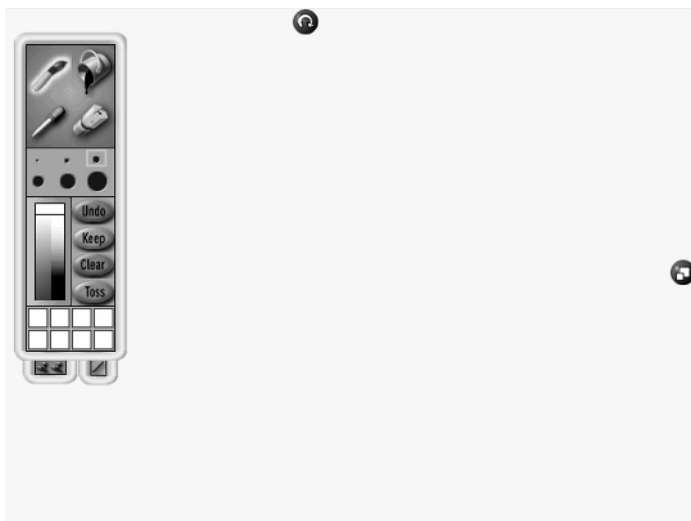


Figure 6-5. The painting editor

Step 2: Draw the new robot graphic. The second step is to draw a new graphic for your

robot. Draw your robot pointing to the right, as shown in Figure 6-6. The painting editor has the usual features of graphics programs, such as selecting the brush size, filling a region, repeating a selected region, and selecting the paint color. The painting tool also has two buttons (shown in Figure 6-7) to rotate and zoom your drawing.

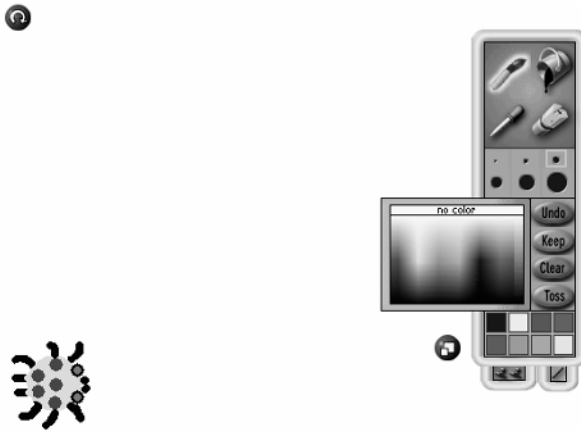


Figure 6-6. *This robot looks like a spotted spider.*



Figure 6-7. *The zoom and rotate buttons*

Step 3: Preserving your graphic. Once you are satisfied with your drawing, you should press the button **keep**. This closes the painting tool. Now your robot looks like the graphic that you created.

Saving and Restoring Graphics

If you have spent a lot of time drawing a robot and you would like to save it for future reference, you can save it to a file. Once it has been saved, you will be able to load it into different environments and share it with your friends. You can begin to build a library of robot graphics over time. Now I will show you how to save and load a graphic. Then I will show how you can associate a graphic with a single robot or even to a class (robot factory), so that all newly created robots will look like the graphics that you have drawn. I will start by showing you how to perform all these manipulations by interacting with the robots directly, and then how to write scripts to do these things automatically.

The “Save Graphics” Handle

To save a graphic, simply click on the blue handle, the one with the file icon (Figure 6-8). I chose the color blue to make you think of a frozen lake: saving the graphic “freezes” your robot’s shape to preserve it. The system will then ask you to give a name to the saved graphic, as shown in Figure 6-9. This operation saves your graphic to a file, in the same folder as the Squeak image, with the name you entered and with the extension `.frm`.

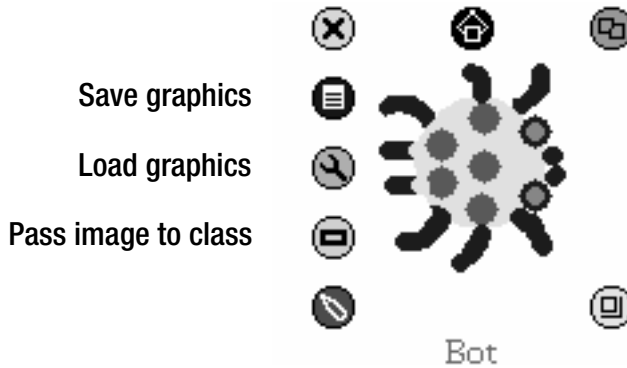


Figure 6-8. *The robot now looks like a spider. It is pointing to the right.*



Figure 6-9. *Clicking on the blue handle produces a prompt for a name.*

You can reverse the operation and load a graphic by clicking on the pink handle in the robot’s halo, the one with an icon that looks like a tool that a robot might use. I chose the color pink to make you think of bringing your robot back to life. When you click on the pink handle, the system asks you for the name of the graphic you want to load. Your robot will take on the appearance of the graphic that you choose.

Retooling the Robot Factory

You have drawn and saved a beautiful spotted spider, and you would like the robot factory to make you a robot with this graphic, but when you tell the `Bot` class to create a new robot, it creates one using the default graphic. In order for the `Bot` class to be able to create your

spider robot, you have to tell your robot to pass its graphic to the class using the message `passImageToClass`. After you have sent this message, if you create a new robot and ask it to look like the image, it will look like the graphic that you just drew.

Another way of obtaining the same result is to send `lookLikeImage` or any of the `lookLike` messages to the `Bot` class itself. Then the class will be configured to create new robots according to the new configuration. For example, if you send the message `lookLikeCircle` to the class `Bot`, all newly created robots will look like a circle. Thus if you want to have the `Bot` class create robots that look like spiders, you have to (1) get a robot, (2) draw the spider or load a previously saved spider graphic, (3) pass the spider image to the class, and (4) tell the class to make robots with the image by sending it the message `lookLikeImage`. Then all your newly created robots will look like a spider, as shown in Figure 6-10.

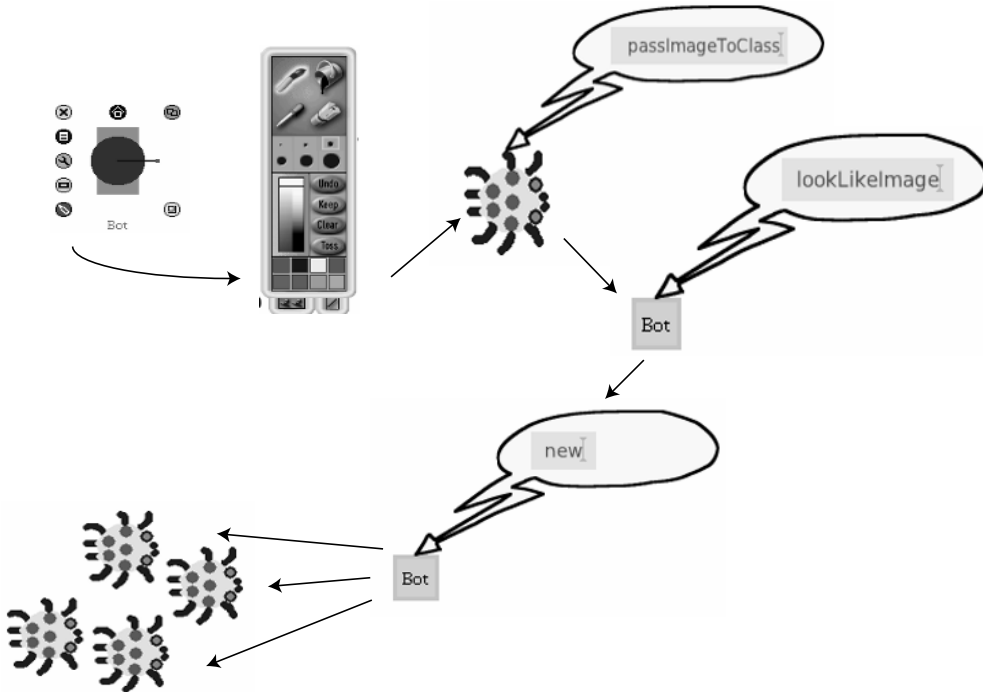


Figure 6-10. Passing an image to the `Bot` class and sending the message `lookLikeImage` results in all newly created robots looking like that image.

Graphics Operations Using Scripts

You can also write a script to load and save graphics and associate them with a single robot or with a class.

Script 6-5 creates two robots and loads a graphic for each of them using two different methods. After `pica` is created, he is sent the message `loadImage`, which results in a prompt to the user for the name of the image to load. Then `berthe` is created, and she is sent the message `loadImage: 'spider'`, which gives her the image contained in the image file `spider.frm`.

Note the important distinction between the messages `loadImage` and `loadImage: 'fileName'`. The first of these has no parameter, and the user is prompted for the name of the graphics file to load. The second message does have a parameter, where here `'fileName'` represents the name of the image file inside single quotation marks (and without the file extension).

You can save the image using the message `saveImage` or `saveImage: 'fileName'`. First `berthe` is sent the message `saveImage`, which prompts the user for the name under which the image is to be saved. Finally, `pica` is sent the message `saveImage: 'spider2'`, which saves his image under the file name `spider2.frm`.

Script 6-5. Two ways of loading and saving robot graphics

```
| pica berthe |
pica := Bot new.
pica loadImage.           "The user is prompted for the name of the image to load"
berthe := Bot new.
berthe loadImage: 'spider' "A parameter gives the name of the file to be loaded"
berthe saveImage.        "The user is prompted to name the saved image file"
pica saveImage: 'spider2' "A parameter gives the name of the saved image"
```

Just as you can load and save graphics associated with an individual robot, you can load and save graphics that are to be associated with a class, such as the `Bot` class. The same messages are used for the class as were used for the individual robots. They are just sent to `Bot` instead of to `pica` or `berthe`. Script 6-6 first associates the image `spider.frm` with the `Bot` class. Then the image is saved under another name, `spiderBot.frm`.

Instead of the methods `loadImage:` and `saveImage:`, you can use `loadImage` and `saveImage` (no colon, no argument), which prompt the user for a file name. The expression `Bot clearImage` resets the `Bot` class to its condition the first time you used it. This restores the default robot image to the class, which means that when you run the script, you can reproduce a predictable scenario.

Script 6-6. Loading and saving a graphic associated with the `Bot` class

```
| pica berthe |
Bot clearImage.          "clears any graphic that was previously associated
                        with the class Bot."
berthe := Bot new.      "berthe looks like a default robot"
Bot loadImage: 'spider'. "The image in spider.frm is associated with the Bot class"
Bot lookLikeImage
pica := Bot new.        "The robot pica looks like a spider"
Bot saveImage: 'spider3' "The spider image is saved under the name spider3.frm"
```

The following scripts (Scripts 6-7 and 6-8) assume that three image files, `luth.frm`, `spider.frm`, and `airplane.frm` are located in the directory containing the Squeak image. These files are included in the distribution of the environment used in this book.

Script 6-7 uses the method `loadImage:` to associate an image with a robot, and the method `lookLikeImage` to instruct a robot to look like the image with which it is associated. After the robot `pica` is created, he is asked to look like his associated image (`pica lookLikeImage`). Since no graphic has been associated with `pica`, asking him to look like an image produces no change. But then the image from the file `luth.frm` is associated with `pica` by sending him the message `loadImage: 'luth'`. Now when `pica` is sent the message `lookLikeImage`, his appearance is changed, and he looks like a luth sea turtle. In the last line of the script, a different image file is associated with `pica`. Once it has been given the message `lookLikeImage`, a robot will look like whatever image is associated with it. Thus when the expression `pica loadImage: 'spider'` is executed, `pica` will look like a spider.

The script continues with a new robot, `berthe`, being created. Since the class `Bot` does not have any image associated with it, `berthe` will have the graphic of a default robot, and if you send her the message `lookLikeImage`, nothing changes, since she does not have an associated image.

Script 6-7. *Changing the image of a robot*

```
| pica berthe |
Bot clearImage.

pica := Bot new.
pica lookLikeImage.
"No image loaded or created, so nothing changes"

pica loadImage: 'luth'.
pica lookLikeImage
"Load an image and ask the robot to look like it"

pica loadImage: 'spider'.
"load another image, and since the message lookLikeImage has already been
sent, pica will look like the new image"

berthe := Bot new.
berthe lookLikeImage.
"When berthe is created, she looks like a default robot, and since no image
has been loaded into the class, the message lookLikeImage causes nothing to
change"
```

Script 6-8 shows how to notify the `Bot` class that all newly created robots should have a particular graphic. In contrast to the situation described in Script 6-7, the message `loadImage: 'fileName'` is sent to the class `Bot` itself and not to a particular robot. Just as a swimmer and a billiards player have different reactions to the word “pool,” different objects and classes have different understandings of the same message. That is because every object or class has its own method that responds to a given message, and these methods may be different for the same message. In the case at hand, `loadImage:` has different behavior depending whether it is received by the `Bot` class or by a robot, which is an *instance* of the class. When received by the

Bot class, the message `loadImage: 'fileName'` leads to the class loading and associating the graphic from the file, so that newly created instances (robots) can use the new graphic. When received by a robot, only the particular robot receiving the message can use this graphic.

Script 6-8. *Associating a graphic with the Bot class*

```
| berthe daly pica yertle |
Bot loadImage: 'spider'.
berthe := Bot new.
berthe lookLikeImage.
"berthe, as an instance of the Bot class, now looks like a spider"

daly := Bot new.
daly lookLikeImage.
"daly also now looks like a spider"

pica := Bot new.
pica loadImage: 'luth'.
pica lookLikeImage.
"But a specific robot can still change its own graphics;
pica now looks like a turtle"

pica getImageFromClass.
"pica gets his image from the Bot class; now he looks like a spider again"

Bot loadImage: 'luth'.
Bot lookLikeImage.
yertle := Bot new.
"Now the class will create robots that look like luth turtles"
```

Script 6-8 starts by loading a new graphic from a file and associating it with the Bot class itself. Then the new robot `berthe` is created, and she is sent the message that tells her to use the new graphic. Creating another robot, `daly`, and sending him the message `lookLikeImage` makes him also look like the image associated with the class.

All robots created can be made to look like a spider. However, a particular robot, such as the robot `pica` in the script, can be given his own image by sending him the message `loadImage: 'fileName'`. The robot's associated image overrides the class image. The message `getImageFromClass` makes it possible to restore the graphic associated with the class. The last sequence of messages shows that we can associate a new graphic to a class, replacing the currently associated image. Sending the message `loadImage: 'fileName'` to the class Bot associates the graphic in the file `fileName.frm` to the class. Then sending the message `lookLikeImage` ensures that newly created robots will by default look like the graphic now associated with the class. Hence the robot `yertle` looks like a turtle.

Summary

Method	Description	Example
lookLikeCircle	Change the shape of the receiver to a circle.	Bot new lookLikeCircle
lookLikeBot	Change the shape of the receiver to a robot.	Bot new lookLikeBot
lookLikeTriangle	Change the shape of the receiver to a triangle.	Bot new lookLikeTriangle
lookLikeImage	Change the appearance of the receiver to the graphic you painted.	Bot new lookLikeImage
lookLikeCircle	Sending to the class results in newly created robots having the shape of a circle.	Bot lookLikeCircle
lookLikeBot	Sending to the class results in newly created robots having the shape of a robot.	Bot lookLikeBot
lookLikeTriangle	Sending to the class results in newly created robots having the shape of a triangle.	Bot lookLikeTriangle
lookLikeImage	Sending to the class results in newly created robots having the shape of the graphic you painted or loaded.	Bot lookLikeImage
loadImage: <i>'fileName'</i>	Load the image file <i>fileName.frm</i> into the class or the robot.	Bot loadImage: 'spider' or berthe loadImage: 'spider'
loadImage	Prompt the user for the name of an image file to be loaded into the class or the robot.	Bot loadImage or berthe loadImage
saveImage: <i>'fileName'</i>	Save the image of the class or the robot to the file named <i>fileName.frm</i> .	Bot saveImage: 'spider' or berthe saveImage: 'spider'
saveImage	Save the image of the class or the robot by prompting the user for a file name.	Bot saveImage or berthe saveImage
penColor: <i>aColor</i>	Change the color of the pen.	berthe penColor: Color blue
penSize: <i>aNumber</i>	Change the size of the pen. The default size is 1.	berthe penSize: 3
color: <i>aColor</i>	Change the color of the receiver to the specified color.	berthe color: Color yellow
extent: <i>aPoint</i>	Change the size of the receiver to dimensions given by <i>aPoint</i> , where <i>aPoint</i> is given by <i>w@h</i> , where <i>w</i> is the width and <i>h</i> is the height.	berthe extent: 80@100
passImageToClass	Pass the graphic of the receiver to the class. After this message, robots created by the class will have as graphic the graphic of the current robot.	berthe passImageToClass
getImageFromClass	Get the graphic of the class. After this message, the receiver will look like the robots that would be created by the class.	berthe getImageFromClass

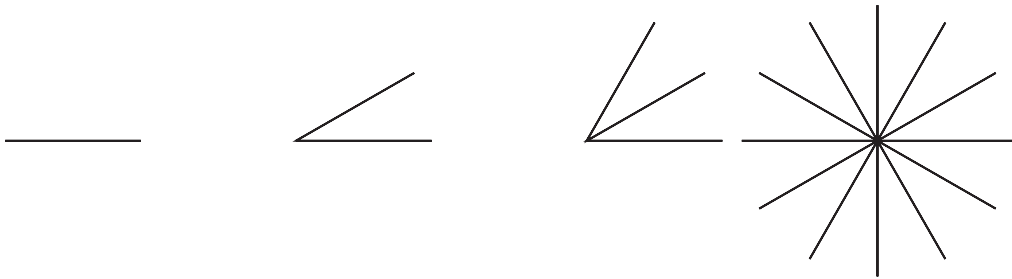
PART 2



Elementary Programming Concepts



Looping



By now, you must think that the job of robot programmer is quite tedious. You probably have a number of ideas for interesting drawings, but you just don't have the heart to write the scripts to draw them, since it appears that the number of lines that you have to type gets larger and larger as the complexity of the drawing increases. In this chapter, you will learn how to use *loops* to reduce the number of expressions given to a robot. Loops allow you to *repeat a sequence of expressions*. With a loop, the script for drawing a hexagon or an octagon is no longer than the script for drawing a square.

A Star Is Born

We would like to instruct a robot to draw a star, similar to the one shown in the picture at the beginning of this chapter. We will instruct pica to draw a star in the following way: starting at what will be the center of the star, draw a line, return to the center, turn through a certain angle, draw another line, and so on until the star is finished. Script 7-1 creates a robot that draws a line of length 70 pixels and then returns to its previous location. Note that after it has returned to its starting point, the robot makes an about-face, so that it is pointing in its original direction.

Script 7-1. *Drawing a line and returning*

```
| pica |
pica := Bot new.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
```

To draw a star, we have to repeat part of Script 7-1 and then instruct the robot to turn through a given angle. Let's draw a six-pointed star, and so the angle will be 60 degrees, since turning 60 degrees each time will result in $360/60 = 6$ branches. Script 7-2 shows how this should be done to obtain a star having 6 branches without using loops.

Script 7-2. *A six-pointed star without loops*

```
| pica |
pica := Bot new.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
```

```

pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.

```

As you can see, after `pica` is created, he repeats the same five lines of code six times (shown in alternating roman and italic type). It seems wasteful to have to type the same code segment over and over. Imagine the length of your script if you wanted a star with 60 branches, like the one shown in Experiment 7-1. What we need is a way of repeating a sequence of expressions.

Loops to the Rescue

The solution to our problem is to use a *loop*. There are different kinds of loops, and the one that I will introduce here allows you to repeat a given sequence of messages a given number of times. The method `timesRepeat:` repeats a sequence of expressions a given number of times, as shown in Script 7-3. This script defines the same star as the one in Script 7-2, but with much less code. Notice that the expressions to be repeated are enclosed in square brackets.

Script 7-3. *Drawing a six-pointed star using a loop*

```

| pica |
pica := Bot new.
6 timesRepeat:
  [ pica go: 70.
    pica turnLeft: 180.
    pica go: 70.
    pica turnLeft: 180.
    pica turnLeft: 60 ]

```

Important! `n timesRepeat: [sequence of expressions]` repeats a sequence of expressions `n` times.

The method `timesRepeat:` allows you to repeat a sequence of expressions, and in Smalltalk, such a sequence of expressions, delimited by square brackets, is called a *block*.

The message `timesRepeat:` is sent to an integer, the number of times the sequence should be repeated. In Script 7-3 the message `timesRepeat: [. . .]` is sent to the integer 6. There is nothing new here; you have a message being sent to an integer when we looked at addition: the second integer was sent to the first, which returned the sum.

Finally, note that the number receiving the message `timesRepeat:` has to be a *whole number*, because in looping as in real life, it is not clear what would be meant by executing a sequence of expressions, say, 0.2785 times.

The argument of `timesRepeat:` is a block, that is, a sequence of expressions surrounded by square brackets. Recall from Chapter 2 that an argument of a message consists of information needed by the receiving object for executing the message. For example, `[pica go: 70. pica turnLeft: 180. pica go: 70.]` is a block consisting of the three expressions `pica go: 70`, `pica turnLeft: 180`, and `pica go: 70`.

Important! The argument of `timesRepeat:` is a *block*, that is, a sequence of expressions surrounded by square brackets.

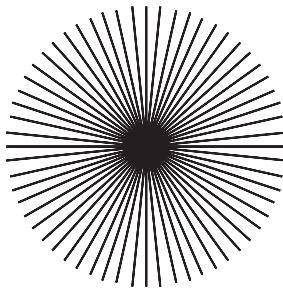
Loops at Work

If you compare Script 7-1 with the expressions in the loop of Script 7-3, you will see that there is one extra expression: `pica turnLeft: 60`, which creates the angle between adjacent branches. There is a simple relationship between the number of branches and the angle through which the robot should turn before drawing the next branch: For a complete star, the relation between the angle and the number of repetitions should be $angle * n = 360$.

To adapt Script 7-3 to draw a star with some other number of branches, you have to change the number of times the loop is repeated by replacing 6 with the appropriate integer. Note that the angle 60 should also be changed accordingly if you want to generate a complete star.

Experiment 7-1 (A Star with Sixty Branches)

Write a script that draws a star with 60 branches.



Code Indentation

Smalltalk code can be laid out in a variety of ways, and its indentation from the left margin has no effect on how the code is executed. We say that indentation has no effect on the syntactic “sense” of the program. However, using clear and consistent indentation helps the reader to understand the code.

I suggest that you follow the convention that was used in Script 7-3 in formatting `timesRepeat: expressions`. The idea is that the repeated block of expressions delimited by the characters `[` and `]` should form a visual and textual rectangle. That is why the block begins with the left bracket on the line following `timesRepeat:` and we align all the expressions inside the block to one tab width. The right bracket at the end indicates that the block is finished. Figure 7-1 should convince you that indented code is easier to read than unindented code.

<pre> pica pica := Bot new. 6 timesRepeat: [pica go: 70. pica turnLeft: 180. pica go: 70. pica turnLeft: 180. pica turnLeft: 60]</pre>	<pre> pica pica := Bot new. 6 timesRepeat: [pica go: 70. pica turnLeft: 180. pica go: 70. pica turnLeft: 180. pica turnLeft: 60]</pre>
---	---

Figure 7-1. *Indenting blocks makes it much easier to identify loops. Left: unindented. Right: indented.*

Code formatting is a topic of endless discussion, because different people like to read their code in different ways. The convention that I am proposing is focused primarily on helping in the identification of repeated expressions.

Drawing Regular Geometric Figures

Many figures can be obtained by simply repeating sequences of messages, such as the square that was drawn in Chapter 4 (repeated here as Script 7-4).

Script 7-4. *Pica's first square*

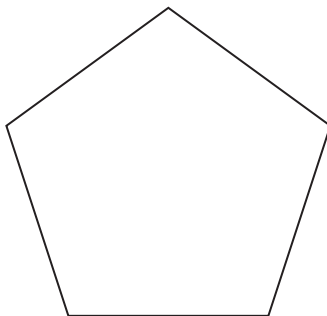
```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
```

Experiment 7-2 (A Square Using a Loop)

Transform Script 7-4 so that it draws the same square using the command `timesRepeat:`. Now you should be able to draw other regular polygons, even those with a large number of sides.

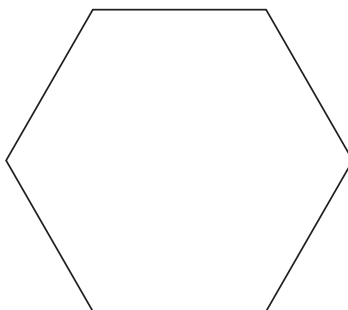
Experiment 7-3 (A Regular Pentagon)

Draw a regular pentagon using the method `timesRepeat`:



Experiment 7-4 (A Regular Hexagon)

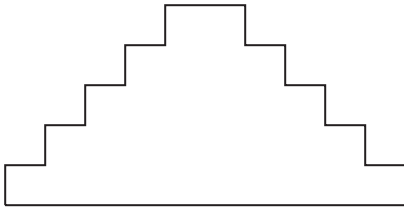
Draw a regular hexagon using the command `timesRepeat`:



Once you have gotten the hang of it, try drawing a regular polygon with a very large number of sides. You may have to reduce the side length to make the figure fit on the screen. When the number of sides is large and the side length is small, the polygon will look like a circle.

Rediscovering the Pyramids

Recall how you coded the outline of the pyramid of Saqqara in Experiment 3-5. You can simplify your code by using a loop, as shown in Script 7-5.

Script 7-5. *A looping pyramid script*

```

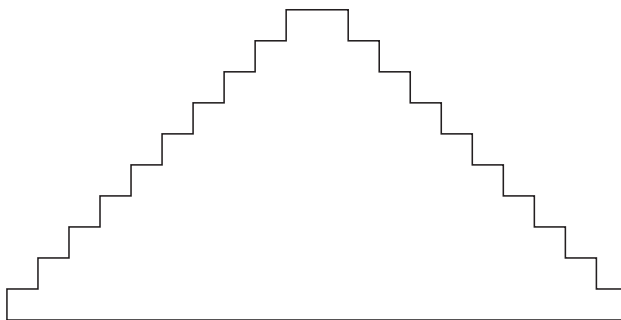
| pica |
pica := Bot new.
5 timesRepeat:
  [pica north.
  pica go: 20.
  pica east.
  pica go: 20].
5 timesRepeat:
  [pica go: 20.
  pica south.
  pica go: 20.
  pica east].
pica west.
pica go: 200.

```

Now you should be able to generate pyramids with an arbitrary number of terraces using the same number of expressions, merely by changing the numbers in the script.

Experiment 7-5 (A Ten-Step Pyramid)

Draw a pyramid with 10 terraces using a variation of Script 7-5.



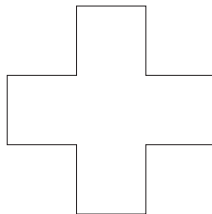
You may now want to generate pyramids with even larger numbers of terraces. The size of the terraces will have to be adjusted if you want them to fit on the screen.

Further Experiments with Loops

As you have seen, generating a step pyramid involves the repetition of a block of code that draws two line segments. Once you have identified the proper repeating element, you can produce complex pictures from elementary drawings through repetition. The following experiments illustrate this principle.

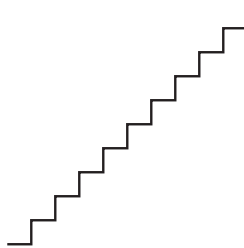
Experiment 7-6 (A Swiss Cross)

Draw the outline of the Swiss cross shown on the right using `turnLeft:` or `turnRight:` and `timesRepeat:`.



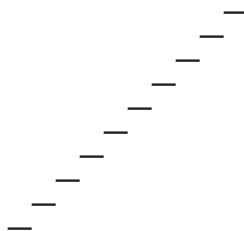
Experiment 7-7 (A Staircase)

Draw the staircase illustrated in the figure.



Experiment 7-8 (A Staircase Without Risers)

Draw the stylized staircase—with treads but without risers—illustrated in the figure.



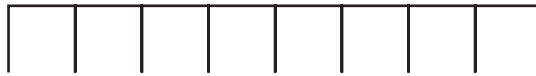
Experiment 7-9 (A Staple)

Draw the illustrated graphical element that looks like a staple.



Experiment 7-10 (A Comb)

Transform the graphical element that you produced in Experiment 7-9 to produce the comb shown in the figure.



Experiment 7-11 (A Ladder)

Transform the graphical element from Experiment 7-9 to produce a ladder.



Experiment 7-12 (Tumbling Squares)

Now that you have mastered loops using `timesRepeat:`, define a loop that draws the tumbling squares illustrated at the start of Chapter 4.

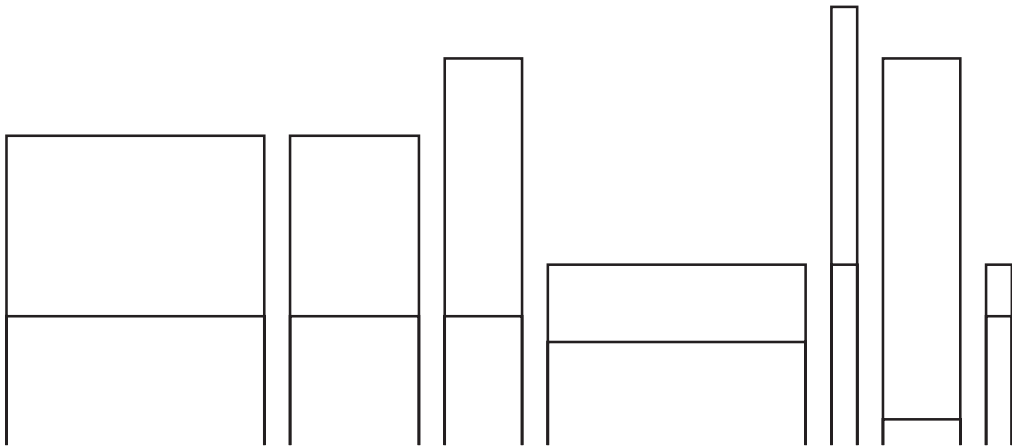
Summary

In this chapter you learned how to program loops using the method `n timesRepeat:`.

Method	Syntax	Description	Example
<code>timesRepeat:</code>	<code>n timesRepeat: [a sequence of expressions]</code>	repeats a sequence of expressions <i>n</i> times	<code>10 timesRepeat: [pica go: 10. pica jump: 10]</code>



Variables



People are always giving names to things. For example, we give names to people, to dogs, and to cars. When we do this, we are *associating* some object, being, or idea with a word or a symbol. Once this association has been made, we may then use the word or symbol to *refer* to or interact with the object associated with it. A name can last a lifetime, or it can be discarded after a short period of time. Sometimes, names refer to other names. For example, an actor generally has several names: a given name, a stage name, and the name of the character that he or she is currently playing on stage or screen. In a programming language, we also need to be able to name things, and *variables* are used for this purpose.

In this chapter, you will learn about variables, which are placeholders for objects, and how variables help to simplify programs. Indeed, variables are often necessary in programming. Finally, as the complexity of the problems that you encounter increases, you will see that you will need to express dependencies between variables. For example, the width of a rectangle might be two-thirds its length. In this chapter I will show you how to use variables to express dependencies between different quantities.

Brought to You by the Letter A

As you did in Chapter 3, suppose you want to use a robot to write letters of the alphabet. The rather primitive letter A that we are going to draw is characterized by its *height*, its *width*, and a *midheight*, which is the height at which the midline of the A should be drawn, as shown in Figure 8-1.

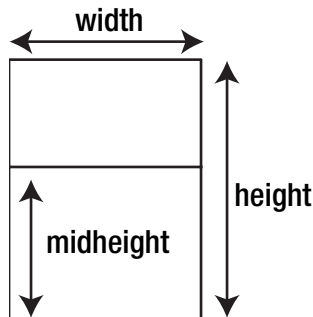


Figure 8-1. The shape of a letter A is characterized by its height, width, and midheight.

Experiment 8-1

Write a script that draws a letter A of height 100 pixels, width 70 pixels, and midheight 60 pixels.

Variations on the Theme of A

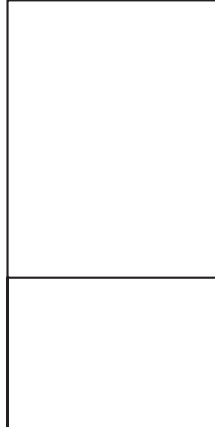
The script you wrote for Experiment 8-1 should resemble Script 8-1.

Script 8-1. *An A for Experiment 8-1*

```
| pica |
pica := Bot new.
pica north.
pica go: 100.
pica east.
pica go: 70.
pica south.
pica go: 100.
pica west.
pica jump: 70.
pica north.
pica go: 60.
pica east.
pica go: 70
```

Experiment 8-2 (*frAnkenstein*)

Modify Script 8-1 to draw a monstrous letter A of height 200 pixels, width 100 pixels, and midheight 70 pixels, as shown in the figure below.



In modifying Script 8-1 for Experiment 8-2, you found that in order to produce an A of a different size, you had to change the numbers that represent the height, the width, and the midheight of the letter *everywhere* they occur and *synchronously*. By synchronously, I mean “in the correct order”; that is, 100 should become 200, 70 should become 100, and 60 should become 70 without any mixups.

Experiment 8-3 (A Variety of A's)

Modify Script 8-1 to draw other A's of different sizes of your choice. Try to reproduce some of the A's that appear in the picture at the start of this chapter.

In doing Experiment 8-3 you undoubtedly quickly came to the conclusion that changing the values of an A's characteristics everywhere is tedious. Moreover, it should be obvious that in making such changes, you run the risk of becoming confused and forgetting to change a value or making a change to an incorrect value. The result can be nothing like what you had in mind for your script. You can imagine that in complex programs, changing values one by one in this way can become highly problematic.

Variables to the Rescue

Making large numbers of changes in producing letters of different sizes and shapes is both tiresome and error-prone, and so we need a solution that will both keep us from mixing up the numbers representing the various characteristics of a letter and enable us to make alterations without having to change all the values everywhere. In fact, we would like to be able to do the following:

- *Declare* the height, width, and midheight of a letter A once for the entire script.
- *Refer* to these values as needed.
- *Change* the values if necessary.

These three things are exactly what a variable allows us to do! Amazing isn't it? A variable is a *name* with which we *associate a value*. We must *declare* it and *associate* a new value with it. Then we can *refer* to a variable and obtain the *value* associated with this variable. It is also possible to *modify* the value associated with a variable and assign it a new value. The value of a variable value can be a number, a collection of objects, or even a robot. We now illustrate how to declare, associate a value, and use a variable.

Important! A variable is a *name* with which we *associate a value*. We *declare* a variable and *associate* a value with it. Then we can *refer* to a variable and obtain its *value*. It is also possible to *modify* the value associated with a variable and associate a new value with it.

Declaring a Variable

Before using a variable, we have to *declare* it; that is, we must tell Squeak the name of the variable that we want to use. We declare variables by enclosing them between vertical bars | |, as shown in the following example, which declares the three variables height, width, and midheight:

```
| height width midheight|
```

To be precise, vertical bars | | declare *temporary* variables, which are variables that exist only during the execution of the script.

Assigning a Value to a Variable

Before using a variable it is almost always necessary to give it a value. Associating a value is called *assigning* a value to a variable. In Smalltalk, the symbol pair := is used in combination to assign a value to a variable. In the following script, after declaring three variables we assign 100 to the variable height, 70 to the variable width, and 60 to the variable midheight. When we assign a value to a variable for the first time, we say that we are *initializing* it:

```
| height width midheight |
height := 100.
width := 70.
midheight := 60
```

Important! The symbol `:=` assigns a value to a variable. For example, `height := 120` assigns the value 120 to the variable `height`, while `length := 120 + 30` assigns the result of the expression `120 + 30`, that is, 150, to the variable `length`.

When we assign a value to a variable for the first time, we say that we are *initializing* it.

Referring to Variables

To refer to the value assigned to a variable—we also say *use* a variable—simply write its name in a script. In the following script, after being *declared* in line 1, the variable `height` is *initialized* with the value 100 in line 3 and *used* in line 5 to tell the created robot to go forward the number of pixels associated with the variable `height`, which here is 100.

```
| pica height |
pica := Bot new.
height := 100
pica north.
pica go: height
```

Important! In general, a variable must be *declared* and *initialized* before being used.

And What About Pica?

You guessed it! `pica` is also a variable. It just happens to be a variable whose value is a robot. Hence, `| pica |` declares a variable named `pica`. The expression `pica := Bot new` initializes the variable with a value, here a new robot. Then we use this robot by sending messages to it via the variable `pica`, for example, `pica go: 100`.

Using Variables

Now let us explore the benefits of using variables, and I will show you some powerful properties that variables possess. In particular, I will show you the power that comes from expressing relationships between variables.

By introducing variables into Script 8-1, we obtain Script 8-2.

Script 8-2. An A with variables

```
| pica height width midheight |
pica := Bot new.
height := 100.           "initializes the variables"
width := 70.
midheight := 60.
pica north.
```

```

pica go: height.           "then we use the variables"
pica east.
pica go: width.
pica south.
pica go: height.
pica west.
pica jump: width.
pica north.
pica go: midheight.
pica east.
pica go: width

```

You will agree that changing variable values once is easier than changing numbers scattered throughout the script. Change some values to convince yourself. You should be able to draw all the A's that appear in the figure at the head of this chapter. Now if you want to change the characteristics of your letter A, you need only reinitialize the variables by changing the values in lines 3, 4, and 5, as shown in Script 8-3. The resulting drawing is presented in Figure 8-2.

Script 8-3. A modified letter A

```

| pica height width midheight |
pica := Bot new.
height := 30.           "initializes the variables"
width := 200.
midheight := 10.
...

```

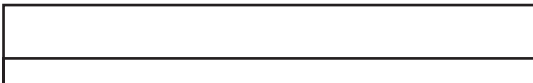


Figure 8-2. A short and stout letter A simply created with $height = 30$, $width = 200$, and $midheight = 10$

Using variables, you can easily create many different letters, and in the future, you will be able to write programs to solve many challenging problems. Let us step back for a moment and consider the power provided by variables.

The Power of Variables

The experiments in the remainder of the chapter illustrate the power of variables. Variables let you name a thing, whether a robot, a length, or practically anything else. Then you can use the names instead of having to repeat the values that you associated with the names. Variables make your scripts much easier to change, since you can simply reinitialize your variables to different values.

In addition, a variable can hold a wide variety of types of values. Up to now, you have assigned robots and numbers to variables, but you can also assign colors (for example, `robotColor := Color yellow`), a sound, or indeed any Squeak object.

Note also that variables make your scripts much more readable and easier to understand. To convince yourself of this, just compare Scripts 8-1 and 8-2. The simple fact of using variables with names such as “width” and “height” helps you to understand how the letter is drawn.

Expressing Relationships Between Variables

In your experiments with the letter A, you probably found some of your letters easier to recognize than others. Letters of the alphabet should generally adhere to certain proportions to keep them readable. In particular, the dimensions that describe a particular letter are not chosen at random, but maintain certain proportions between them.

In our simple letter A, let us decide that the width should be two-thirds of the height, and the midheight should be three-fifths of the height. We can express these relationships using variables, as shown in Script 8-4. As you can see, the value of a variable does not have to be a simple number, but can be the result of a complex computation.

Script 8-4. *Relations between variables: a first approximation*

```
| pica height width midheight |
pica := Bot new.
height := 120.
width := 120 * 2 / 3.
midheight := 120 * 3 / 5.
...
```

In looking over Script 8-4, you will soon realize that it is not optimal. The relationships between the variables are expressed not between the variables themselves, but in terms of the value 120 (in lines 3, 4, and 5). This value would have to be changed manually whenever you wanted to produce a different letter A with the same proportions. You want to be able to change the value of `height` and have the values of `width` and `midheight` change automatically. The solution is to use the variable `height` instead of 120, as shown in Script 8-5. In this script, the values of the variables `width` and `midheight` truly depend on the value of `height`. What makes this work is that the value of a variable can be expressed in terms of other variables. The expression `width := height * 2 / 3` expresses that the width of the letter is equal to two-thirds of its height.

Script 8-5. *Relations between variables: The variables width and midheight depend on height.*

```
| pica height width midheight |
pica := Bot new.
height := 120.
width := height * 2 / 3.
midheight := height * 3 / 5.
pica north.
...
```

Initialize Before Using!

The only constraint that you have to consider in expressing relationships between variables is that a variable used in the definition of another variable should have a value. For example, in Script 8-5, the variable `height` has its initialized value 120, which is used by `width` and `midheight` in computing their initialized values. To see what can go wrong, in Script 8-6 the variable `height` has not been initialized, and so when an attempt is made to initialize the variable `width` to `height * 2/3`, this leads to an error, because there is no value of `height` to be used in the computation. I will have more to say about errors in Chapter 15.

Script 8-6. Problematic `width` initialization

```
| height width midheight |  
width := height * 2 / 3.  
height := 120.  
midheight := height * 3 / 5.
```

Experimenting with Variables

The experiments that follow will help you to gain experience with variables.

Experiment 8-4 (Golden Rectangle)

A golden rectangle is a rectangle one of whose sides is approximately 1.6 times the length of the other. The number 1.6 is an approximation of the “golden ratio”: the number $\frac{1 + \sqrt{5}}{2}$. A nice property of such a rectangle is that if you cut off a square inside the rectangle, as shown in the figure below, then the part of the rectangle left over is again a golden rectangle. You can then cut a square off of this smaller golden rectangle and obtain an even smaller golden rectangle, and so on ad infinitum. The dimensions of a golden rectangle are pleasing to the eye, and since ancient times, artists and architects have used the golden ratio in their work. Write a script that draws a golden rectangle. You can express the number in Smalltalk as `1 + 5 sqrt / 2`.



Experiment 8-5 (Scripts That Don't Work)

Explain why neither of the following scripts is able to draw a letter A of height 120 pixels.

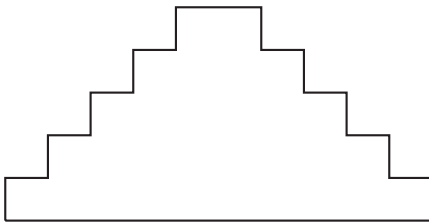
```
| pica height |
pica := Bot new.
height := 120.
pica north.
pica go: 100.
pica east.
pica go: 70.
pica south.
pica go: 100.
pica west.
pica jump: 70.
pica north.
pica jump: 50.
pica east.
pica go: 70
```

```
| pica height |
pica := Bot new.
pica north.
pica go: height.
pica east.
pica go: 70.
pica south.
pica go: height.
pica west.
pica jump: 70.
pica north.
pica jump: 50.
pica east.
pica go: 70
```

The Pyramids Rediscovered

In Script 7-5, in Chapter 7, we defined the outline of the step pyramid of Saqqara as in Script 8-7.

Script 8-7. *The pyramid of Saqqara*



```
| pica |
pica := Bot new.
5 timesRepeat:
  [ pica north.
    pica go: 20.
    pica east.
    pica go: 20 ].
5 timesRepeat:
  [ pica go: 20.
    pica south.
    pica go: 20.
    pica east ].
pica west.
pica go: 200.
```

Experiment 8-6 (A Pyramid with a Variable Number of Terraces)

Modify Script 7-5, introducing the variable `terraceNumber` to represent the number of terraces that the pyramid should have.

Experiment 8-7 (A Pyramid with a Variable Terrace Size)

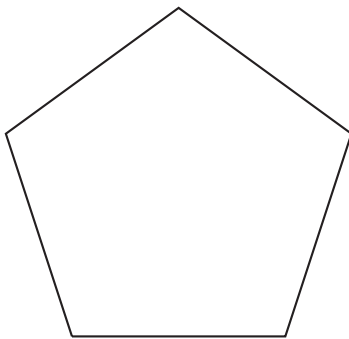
Modify the script from Experiment 8-6 by introducing the variable `terraceSize` to represent the size of a terrace.

Automated Polygons Using Variables

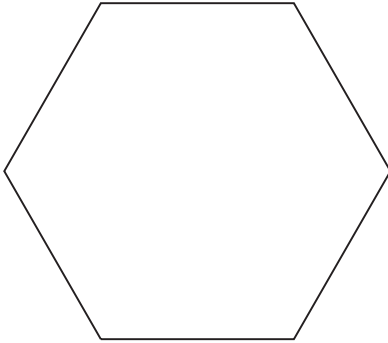
The use of variables greatly simplifies the definition of scripts in which some of the *variables* depend on other variables. In this section, you will see how the use of variables provides great leverage in dealing with loops. Chapter 10 will go more deeply into the power that the combination of variables and loops can give to your programs.

Let us look again for a moment at Experiments 7-3 and 7-4, in which a Bot was asked to draw a regular pentagon and a regular hexagon. The requisite code appears here as Scripts 8-8 and 8-9.

Script 8-8. *A regular pentagon*



```
| pica |  
pica := Bot new.  
5 timesRepeat:  
  [ pica go: 100.  
    pica turnLeft: 72 ]
```

Script 8-9. *A regular hexagon*

```
| pica |
pica := Bot new.
6 timesRepeat:
  [ pica go: 100.
    pica turnLeft: 60 ]
```

In order to transform the first script into the second, you must change the number of sides (let us call it s) *and* the magnitude of the turn (let us call it T) such that the product $s \times T$ is equal to 360. Wouldn't it be nice if we could write a script in which we would have to change only a single number, let us say the number of sides, since this is the easiest parameter to choose? This can be done by introducing variables. Try to come up with your own solution.

Script 8-10 solves the problem. It makes it possible to draw a regular polygon with any number of sides by changing a single number. Try it before I discuss it further.

Script 8-10. *Drawing a regular polygon*

```
| pica sides angle |
pica := Bot new.
sides := 6.
angle := 360 / sides.
sides timesRepeat:
  [ pica go: 100.
    pica turnLeft: angle ]
```

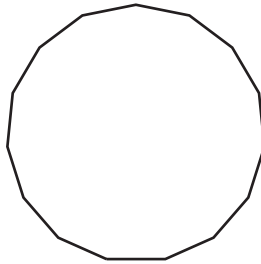
This script introduces two new variables, `sides` and `angle`, which are used to hold the number of sides and the size of the angle. Then, the expression `sides := 6` assigns the value 6 to the variable `sides`, and the expression `angle := 360 / sides` assigns a value to the variable `angle`, which is the result of 360 divided by the value held in the variable `sides`. The value of the variable `angle` is then used as the argument of the command `turnLeft`: given to the robot in the repeating block.

Regular Polygons with Fixed Sizes

You will notice that if Script 8-10 is executed with a large number of sides, the resulting figure does not fit on the screen. The next experiment asks you to fix this problem by reducing the length of the sides as the number of sides increases.

Experiment 8-8 (Controlling the Sides of the Polygon)

Modify Script 8-10 so that the size of the regular polygon stays roughly constant as the number of sides changes. Hint: Introduce a variable `totalLength` that is set to a fixed length, and let each side of your polygon have length equal to `totalLength` divided by the number of sides.



Summary

- A variable is a *name* with which we *associate a value*. We must *declare* it and *assign* a value to it. Then we can *refer* to a variable and obtain the *value* associated with this variable. It is also possible to *modify* the value associated with a variable and assign a new value to it.
- A variable can be used at any place where its value can be used.
- The first time that we assign a value to a variable, we say that we are *initializing* it.
- The symbol `:=` assigns a value to a variable. For example, `height := 120` assigns the value 120 to the variable `height`, while `length := 120 + 30` assigns the result of the expression `120 + 30`, that is, 150, to the variable `length`.
- A variable must generally be *declared* and *initialized* before being used.

Use of Variable	Syntax	Description	Example
variable declaration	<i>variablename</i>	Declares <i>variablename</i> as a variable	pica height
variable assignment	<i>variablename</i> := <i>expression</i>	Assigns the value of <i>expression</i> to the variable <i>variablename</i>	length := 40 length := 30 + 20
		Using a variable's value in an expression	pica go: length
		Changing the value of a variable using its current value	length := length + 10



Digging Deeper into Variables

In the previous chapter I introduced variables. In this chapter I am going to delve a bit deeper into the subject so that you can learn more about how variables are used. Since this chapter is a bit technical in nature, you might want to omit it on a first reading.

Before illustrating in detail how variables work, I want to stress again the importance of choosing good names for variables.

Naming Variables

You are free to choose practically any name for a variable. However, giving your variables meaningful names is very important, because doing so will help you both in writing your programs and in understanding the programs that you have written. To illustrate this point, read Script 9-1, which is in fact Script 8-10 rewritten using meaningless variable names.

Script 9-1. *Meaningless variable names make a program difficult to understand.*

```
| x y z |
x := Bot new.
y := 6.
z := 360 / y.
y timesRepeat:
    [ x go: 100.
      x turnLeft: z ].
```

As you may discover by trying it out, Script 9-1 is perfectly correct, and Squeak can execute it without any problem. But I am sure that you know which of the two scripts 8-10 and 9-1 is more understandable.

In Smalltalk, the name of a variable can be any sequence of alphabetic and numeric (*alphanumeric*) characters beginning with a lowercase letter. It is customary to use long variable names that clearly indicate the function of the variable in the program. Doing so helps you and other programmers to understand your scripts more easily.

Being able to understand what a program does is very important, since as you will see later, a program usually involves a combination of many scripts.¹ When the time comes that you need to understand a script written by someone else, or even one written by yourself, but perhaps many months ago, you will be glad that you adopted the habit of choosing meaningful variable names.

Now that you have been convinced of the importance of choosing meaningful names for variables, I will discuss variables in detail.

Variables as Boxes

Variables are placeholders that refer to objects. A common way to explain the notion of a variable is to use a graphical notation in which variables are represented as boxes. Let us illustrate this idea in Script 9-2 and Figure 9-1.

In Script 9-2 (step a in the script and in the figure), two variables, `pica` and `pablo`, are declared. Then in step (b) we create a robot and assign it to the variable `pica`; that is, the variable `pica` now refers to the newly created robot. Then in (c) the variable `pablo` is assigned the value of the variable `pica`, and therefore `pablo` now points to the same object as the variable `pica`, that is, to the newly created robot. When we send a message using either of the two variables, we are actually sending that message to the same object, namely the robot created in step (b), since both variables refer to the same object. Therefore, in (d), the message sent to

1. This is actually a simplification. Soon, you will learn about *methods*, which are the true building blocks of programming with objects.

pica causes him to move 100 pixels, while the message in (e), which addresses pablo, causes the *same robot* to change its color to yellow.

Another way of saying this is that the robot has two names: pica and pablo. It is just as if the artist Pablo Picasso's mother had said, "Picasso, come here" (pica go: 100), and then said, "All right, Pablo, here is your yellow shirt. Put it on" (pablo color: yellow).

Script 9-2. *Two variables point to the same robot.*

- (a) | pica pablo |
- (b) pica:= Bot new.
- (c) pablo := pica.
- (d) pica go: 100.
- (e) pablo color: Color yellow.

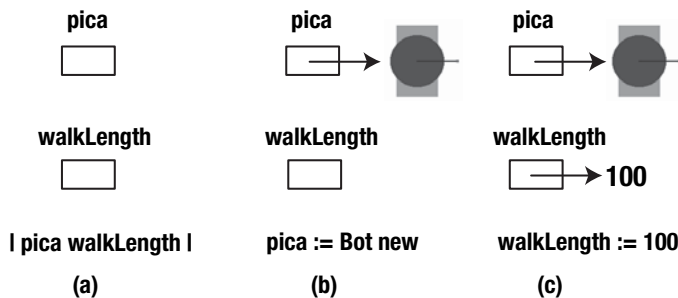


Figure 9-1. (a) Two variables, pica and pablo, are declared. (b) A robot is created and the variable pica is associated with this new object. (c) The variable pablo is assigned the value of the variable pica, and therefore pablo now points to the same object as pica. (d) When we send the message go: 100 using pica, the robot moves. (e) When we send the message color: Color yellow to pablo, the same robot changes color. In sum, if we send a message to either of the two variables, we are actually sending that message to the same object.

Assignment: The Right and Left Parts of :=

There are two very different ways that a variable name is used in a script. Sometimes, the name is used to refer to its value, as in expressions such as walkLength + 100 and pica go: 100. At other times, the variable name is used to refer to the placeholder itself in order to initialize it or change its value, as in walkLength := 100 and pica := Bot new.

The key thing to understand about variables is that using a variable name always refers to the value associated with the variable, *except in the case that the variable is on the left-hand side of an assignment expression*, that is, to the left of the symbol :=. In this one case, the variable name represents the placeholder itself and not the value of the variable. Another way of saying this is that the value of a variable is always *read*, except when it appears to the left of :=, in which case it is *written*, that is, changed. Script 9-3 shows an example.

Script 9-3. *The variable walkLength is written in line 3 and then read in line 4.*

```
(1) | walkLength pica |
(2) pica := Bot new.
(3) walkLength := 100.
(4) walkLength + 150.
(5) pica go: walkLength
```

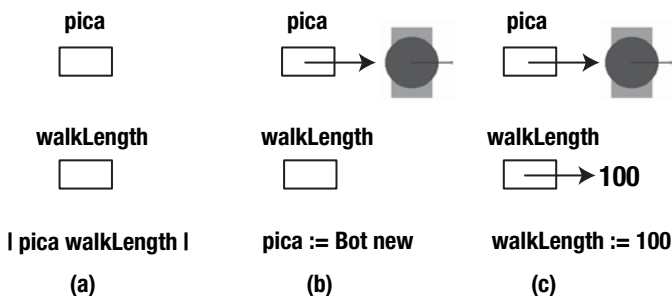
In line 3 of Script 9-3, the variable name `walkLength` appears to the left of `:=`, so it refers to the placeholder, and the value 100 is assigned to the variable `walkLength`. After line 3 has been executed, the variable `walkLength` refers to the number 100. In line 4, `walkLength + 150` is not part of an assignment expression, and so the variable name refers to the variable's value. (Note that in line 4, an addition is carried out, but the result of the addition is not used. Therefore, this line does not do anything and could be removed.) In line 5, both variables `pica` and `walkLength` are used to refer to their values, that is, the objects to which they refer. Therefore, the variable `walkLength` here refers to its value 100, while `pica` refers to the robot created earlier in the script. Thus line (5) has the effect that the message `go: 100` is sent to the robot created the line 2.

Important! A variable is a placeholder for a value, that is, an object. Using a variable returns its value except when the variable is on the left-hand side of an assignment expression, that is, to the left of the symbol `:=`. In such a case, the value of the variable is changed to the value of the expression on the right-hand side of the assignment expression. For example, `walkLength + 150` returns 150 added to the *value* of the variable `walkLength`, while `walkLength := 100` *changes* the value of `walkLength` to 100.

Analyzing Some Simple Scripts

To understand better how variables are manipulated, I am going to describe a series of scripts. First, read the script and guess what it does; then evaluate the script carefully to determine its result. I suggest that you to draw a box representation if you think it would be helpful, and check your drawing against the one shown in the figure. As you will see, I give a small explanation of each script. Note that explanations from one script are not repeated in subsequent scripts, so go through them in order, and look back at the previous scripts for clarification of any points that are confusing.

Script 9-4. *Two variables are declared and initialized, and then their values are used.*



```
| pica walkLength |
pica := Bot new.
walkLength := 100.
pica go: walkLength
```

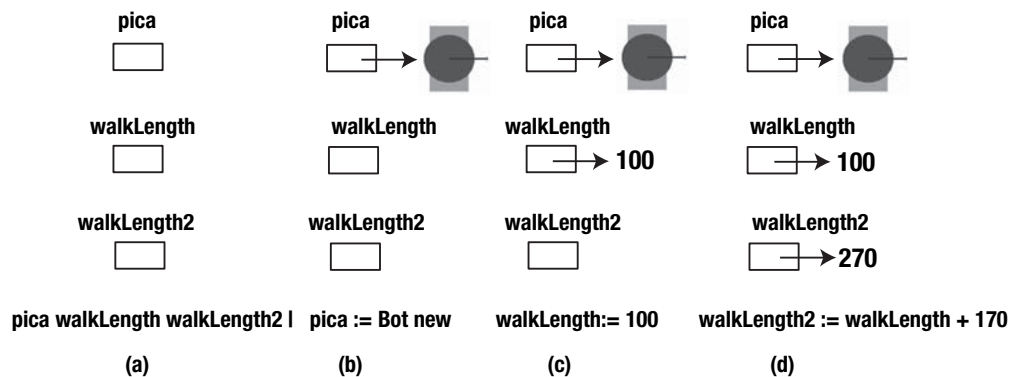
In script 9-4, the variables `pica` and `walkLength` are declared. A new robot is created and assigned to the variable `pica`. Then 100 is assigned to the variable `walkLength`. The robot (value assigned to the variable `pica`) receives the message `go:` with the value of the variable `walkLength` as argument, which in this case is 100.

Script 9-5. Like Script 9-4, except that a variable is used as part of an expression

```
| pica walkLength |
pica := Bot new.
walkLength := 100.
pica go: walkLength + 170.
```

In script 9-5, to determine the number of pixels that the robot should move forward, the expression `walkLength + 170` is evaluated. Since `walkLength` was initialized to the value 100 and its value was not changed thereafter, the value of `walkLength` is 100. Therefore, `walkLength + 170` has the value 270, and the robot moves forward 270 pixels. The expression `pica go: walkLength + 170` in Script 9-5 is equivalent to the expression `pica go: walkLength2` of Script 9-6. Indeed, the value of the variable `walkLength2` is the value of the variable `walkLength` plus 170.

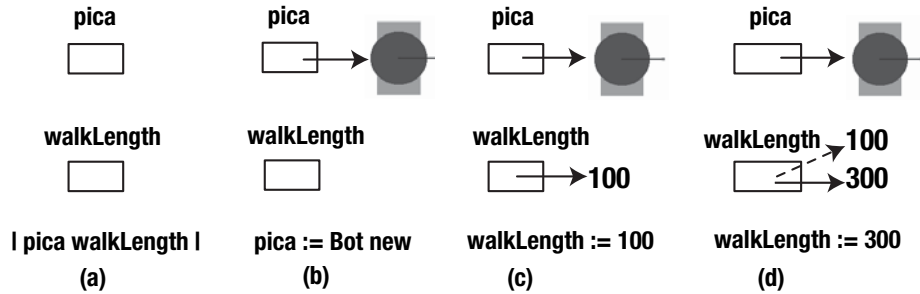
Script 9-6. Using a new variable to hold the value of the length of `pica`'s walk



```
| pica walkLength walkLength2 |
pica := Bot new.
walkLength := 100.
walkLength2 := walkLength + 170.
pica go: walkLength2.
```

The value of a variable can indeed be changed using `:=`. In Script 9-7, first the variable `walkLength` is declared, then we assign 100 to it, and then we assign 300 to it (the dashed arrow in the figure pointing to 100 indicates that the variable no longer points to 100). When the variable is then used in the last expression of the script, its value is 300. As a result, the robot moves forward 300 pixels.

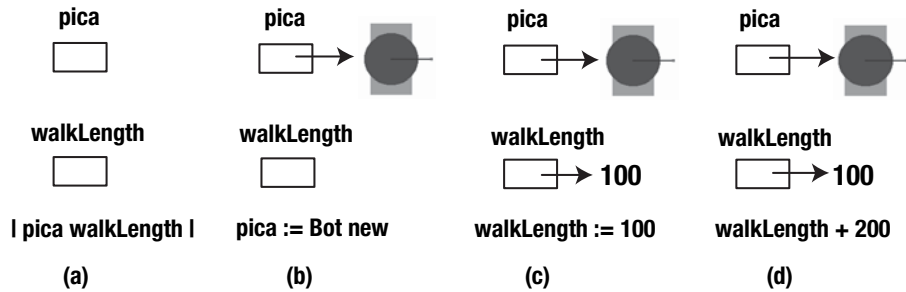
Script 9-7 *Changing walkLength twice*



```
| pica walkLength |
pica := Bot new.
walkLength := 100.
walkLength := 300.
pica go: walkLength
```

Script 9-8 shows that using the value of a variable in any way other than assigning it a new value has no effect on the variable's value. The only way to change the value of a variable is with an assignment expression. In Script 9-8, the variable `walkLength` is initialized with the value 100. Then the value of `walkLength`, here 100, is added to 200, but no assignment expression is involved, and so the variable's value is not modified. And so when the value of `walkLength`, which here is 100, is used in the last statement to specify how far the robot should move forward, the robot moves 100 pixels.

Script 9-8. *Using a variable without assigning it a value has no effect on its value.*

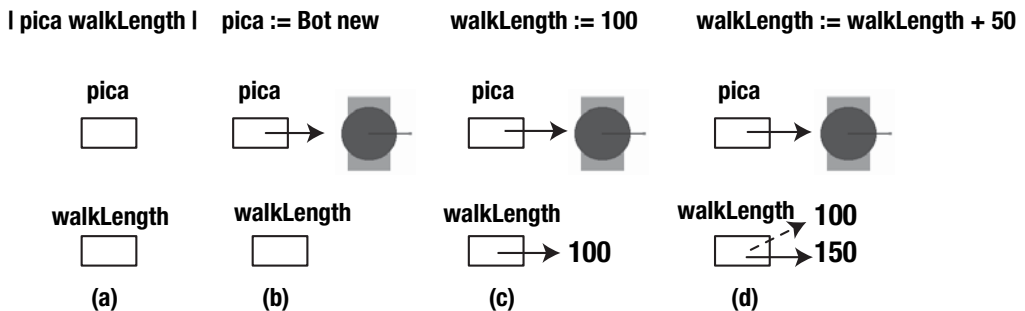


```
| walkLength pica |
pica := Bot new.
walkLength := 100.
walkLength + 200.
pica go: walkLength
```

In Script 9-9, the variable `walkLength` is initialized to 100. Then its value is changed to the value of the expression `walkLength + 50`. At this point, the value of `walkLength` is 100, so the expression `walkLength + 50` returns 150, and then the value 150 is assigned to the variable `walkLength`. So in the last step, the robot moves forward 150 pixels.

It is important to note here that in the expression `walkLength := walkLength + 50`, the variable name `walkLength` is used in *two different ways*: first, the expression to the right of `:=` is evaluated, in which `walkLength` represents a value, and then the result of that evaluation is assigned to the variable `walkLength` to the left of `:=`, which on this side represents the variable as a placeholder.

Script 9-9. *The value of the variable `walkLength` is used to define the variable itself.*



```
| pica walkLength |
pica := Bot new.
walkLength := 100.
walkLength := walkLength + 50.
pica go: walkLength
```

In Script 9-10, the variable `walkLength` is initialized to 150. Then the value of `walkLength` is reassigned to refer to the value of the expression `walkLength + walkLength`. In computing the value of the expression `walkLength + walkLength`, the value of `walkLength` is 150. Therefore, the expression returns 300, which becomes the new value of the variable `walkLength`. And so the robot moves forward 300 pixels.

Script 9-10

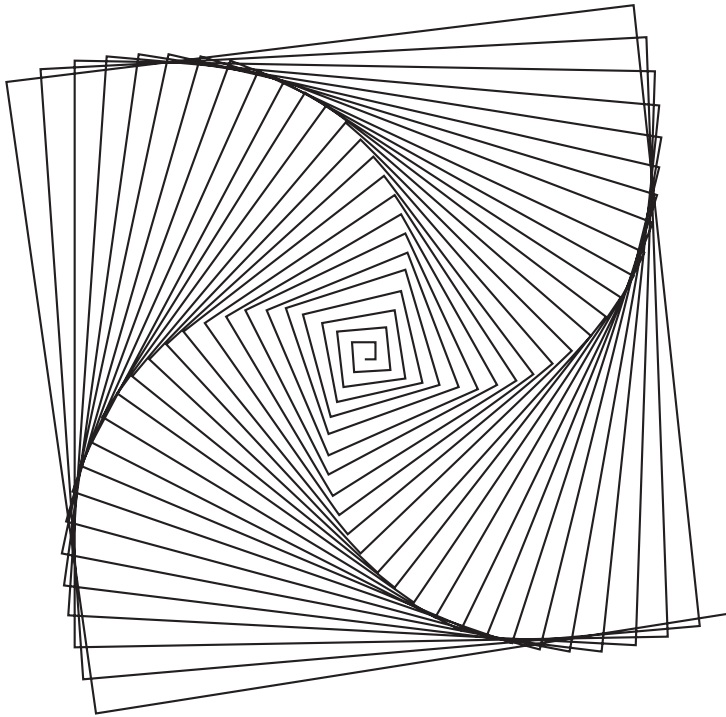
```
| pica walkLength |
pica := Bot new.
walkLength := 150.
walkLength := walkLength + walkLength.
pica go: walkLength
```

Summary

- A variable is an object that serves as a placeholder for a value. You can think of a variable as a box referring to an object.
- Using a variable returns its value except when the variable is on the left of an assignment expression `:=`. In such a case its value changes to become the value of the expression on the right of the assignment expression `:=`. For example, `walkLength + 100` returns 100 added to the value of the variable `walkLength`. On the other hand, `walkLength := 100` changes the value of `walkLength` to be 100.



Loops and Variables



In this chapter I will show you how to use variables and loops in combination. We will begin by analyzing a simple problem that illustrates the need for using variables with loops. Then you will experiment with some other problems.

A Bizarre Staircase

Try to program a robot to draw the strange Staircase shown in Figure 10-1. All of the risers have the same height, but the treads get longer and longer as you climb the staircase. One way to start is to write a script for a normal stairway and then modify it. You will need to solve the problem of making each tread longer than the previous one.

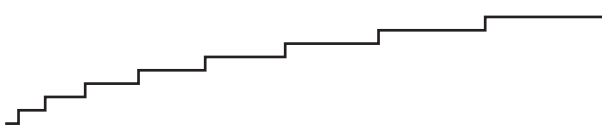


Figure 10-1. *Pica draws a bizarre staircase.*

A simple solution is shown in Script 10-1, where the length of each tread grows by 10 pixels. However, such a solution is not satisfactory for two reasons: First, you have to compute the length of each tread manually. And second, you have to repeat an almost identical sequence of message sends over and over.

Script 10-1. *Pica draws the bizarre staircase.*

```
| pica |  
pica := Bot new.  
pica go: 10.  
pica turnLeft: 90.  
pica go: 5.  
pica turnRight: 90.  
pica go: 20.  
pica turnLeft: 90.  
pica go: 5.  
pica turnRight: 90.  
pica go: 30.  
pica turnLeft: 90.  
pica go: 5.  
pica turnRight: 90.  
pica go: 40.  
pica turnLeft: 90.  
pica go: 5.  
pica turnRight: 90.  
...
```

We would like to be able to use the power of variables (to automate the increasing tread length) combined with the power of loops (so that we don't have to type so much code). To avoid repeating the sequence of message sends you can use the `timesRepeat:` message. And as for using variables, the key is to be found in an examination of Script 10-1, where you will see that the length of each tread after the first is the length of the previous tread plus 10 pixels. After all, $20 = 10 + 10$, $30 = 20 + 10$, $40 = 30 + 10$, and so on, as shown in Figure 10-2.

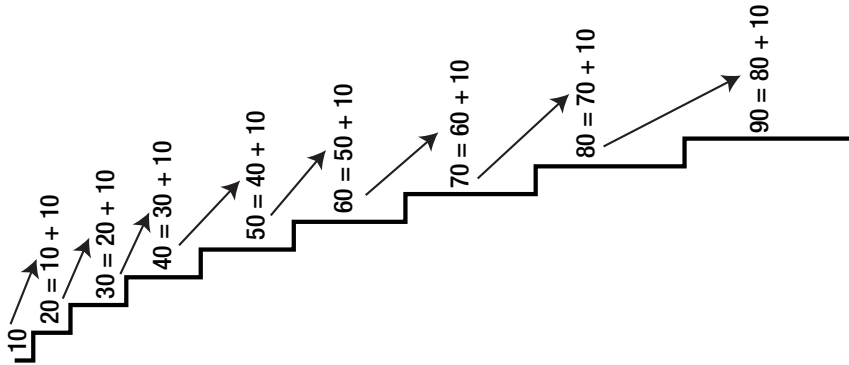


Figure 10-2. *The length of a tread is the length of the previous tread plus 10 pixels.*

Let us use the variable `treadLength` to represent the length of a tread. Then, once `treadLength` has been initialized to the length of the first tread, the expression `treadLength := treadLength + 10` will set the length of the *next* tread to be the value of the *current* tread increased by 10. The result is that if `treadLength` is initialized to 10, and the first tread is drawn, that tread will of course have length 10. Then, after the expression `treadLength := treadLength + 10` is executed, the next time a tread is drawn it will have length 20. And after the expression is executed again, the next tread will have length 30, and so on.

Let's combine everything! We will start with the script of a normal staircase (Script 10-2). Then, in Script 10-3, we obtain the same staircase, but using the variable `treadLength`. Then in Script 10-4, we add the line `treadLength := treadLength + 10` to change the `treadLength` value in each step of the loop, and thus we finally obtain the stairway we want.

Script 10-2. *A stairway with normal steps*

```
| pica |
pica := Bot new.
10 timesRepeat:
    [ pica go: 10.
      pica turnLeft: 90.
      pica go: 5.
      pica turnRight: 90 ]
```

Script 10-3. *A stairway with normal steps using the variable `treadLength`*

```
| pica treadLength |
pica := Bot new.
treadLength := 10.
10 timesRepeat:
    [ pica go: treadLength.
      pica turnLeft: 90.
      pica go: 5.
      pica turnRight: 90 ]
```

Script 10-4. *The solution: increasing the variable `treadLength` each time through the loop produces the bizarre staircase.*

```
| pica treadLength |
pica := Bot new.
treadLength := 10.
10 timesRepeat:
    [ pica go: treadLength.
      pica turnLeft: 90.
      pica go: 5.
      pica turnRight: 90.
      treadLength := treadLength + 10 ]
```

Let's look more closely at the sequence of expressions in the loop in Script 10-4. The first expression draws a tread by making the robot go forward a distance given by the value of the variable `treadLength` (which has been initialized to 10 for the first time through the loop). Then the robot turns, draws a riser (straight up), and turns again. Then the value of the variable `treadLength` is increased by 10 and the loop restarts, but now the variable `treadLength` has a new, larger, value (20 for the second repetition). The whole process is executed 10 times.

The expression `treadLength := treadLength + 10` is absolutely necessary. Without it, the value of the variable would never change.

Experiment 10-1 (Placement of the Increment in the Loop)

Try changing the last line of the loop; for example, `treadLength := treadLength + 15`. Then try moving the line to different places in the loop. Can you explain what happens when you move the last line of the loop to the beginning of the loop?

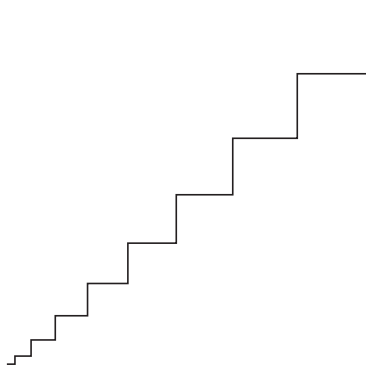
If you are still uncertain about what is going on in Script 10-4, I suggest that you think carefully about the value of the variable `treadLength`, especially at the beginning and end of the loop. Figure out the value of `treadLength` in each expression for three repetitions of the loop. If necessary, read Chapter 8 again.

Practice with Loops and Variables: Mazes, Spirals, and More

Let's see how combining variables and loops can help you to solve some other problems.

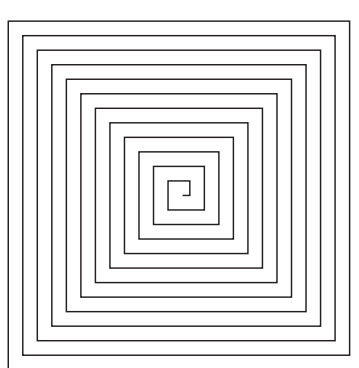
Experiment 10-2 (Another Bizarre Staircase)

Modify Script 10-4 to produce the picture shown below, which represents a staircase in which both the treads and the risers grow in size.



Experiment 10-3 (A Simple Maze)

Write a script that reproduces the drawing shown below. In addition, by changing the angle through which the robot turns, you should be able to re-create the picture shown at the beginning of this chapter, as well as the spiral shown in Figure 10-3.



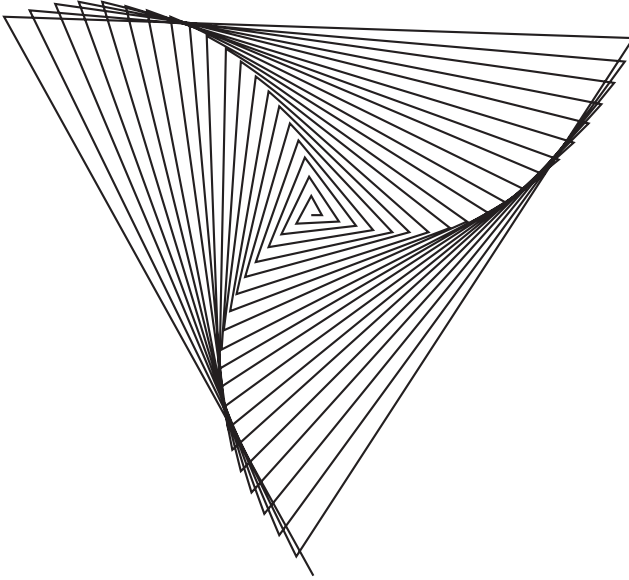
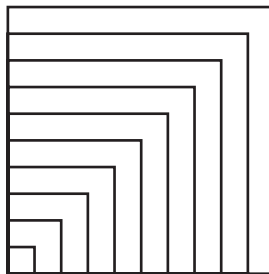


Figure 10-3. *A nice spiral*

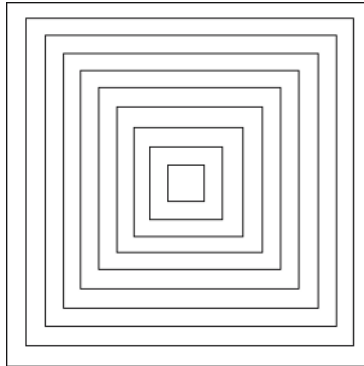
Experiment 10-4 (Russian Squares)

Draw the nested squares of different sizes as shown in the figure below. You might begin by defining a loop that draws the same square ten times. Then introduce a variable `sideLength` to represent the side length of a square, and finally, make the side length grow each time through your loop. As an additional challenge, you might try to write a new script that draws the same figure without the robot having either to jump or draw any line twice.



Experiment 10-5 (A Long Corridor)

The concentric (having a common center) squares of different sizes shown in the figure below represent a long corridor, which seems to get smaller as you look into the distance. Again start by drawing a square, but this time draw it starting from its center (you will need a `jump` message), so that when you change the square's size, the next square will automatically be drawn concentric to the first one. Now define your square inside a loop, and then introduce a variable `sideLength` representing the side length of the square. Finally, make the square grow each time through your loop.



Some Important Points for Using Variables and Loops

Now that you have seen the overall process of combining loops and variables and have experimented a bit, I would like to stress some important points. Script 10-5 shows a typical situation in outline form: First a variable is declared. Then it is initialized. Inside the loop, the variable is used to perform some computations, and then its value is changed for the next pass through the loop.

Script 10-5. *An outline script showing the use of a variable in a loop*

```
| treadLength pica |                               "variable declaration"
...
treadLength := 10.                                "initialization of the variable"
...
10 timesRepeat:
  [ pica go: treadLength.                          "variable use"
    ...
    treadLength := treadLength + 10 ]             "variable change of value"
```

Variable Initialization

When you introduce a variable in a loop, you have to pay special attention to the first value of the variable, that is, the value assigned to the variable when it is initialized. Keep in mind that a variable cannot be used until it has been initialized. Normally, variable initialization is done outside of the loop, for otherwise, the variable's value would be reinitialized at each step of the loop, with the result that the value of the variable would not change.

Using and Changing the Value of a Variable

Inside the loop, the variable's value is often used to perform a variety of computations, such as to compute the values of other variables or perhaps to tell a robot how far to travel. Then the value of the variable is eventually changed. In the stairway example, the expression `treadLength := treadLength + 10` increases the value of the tread length based on its preceding value. What is important to understand is that the new value assigned to the variable will be its value for the next step of the loop, as illustrated in Figure 10-4.

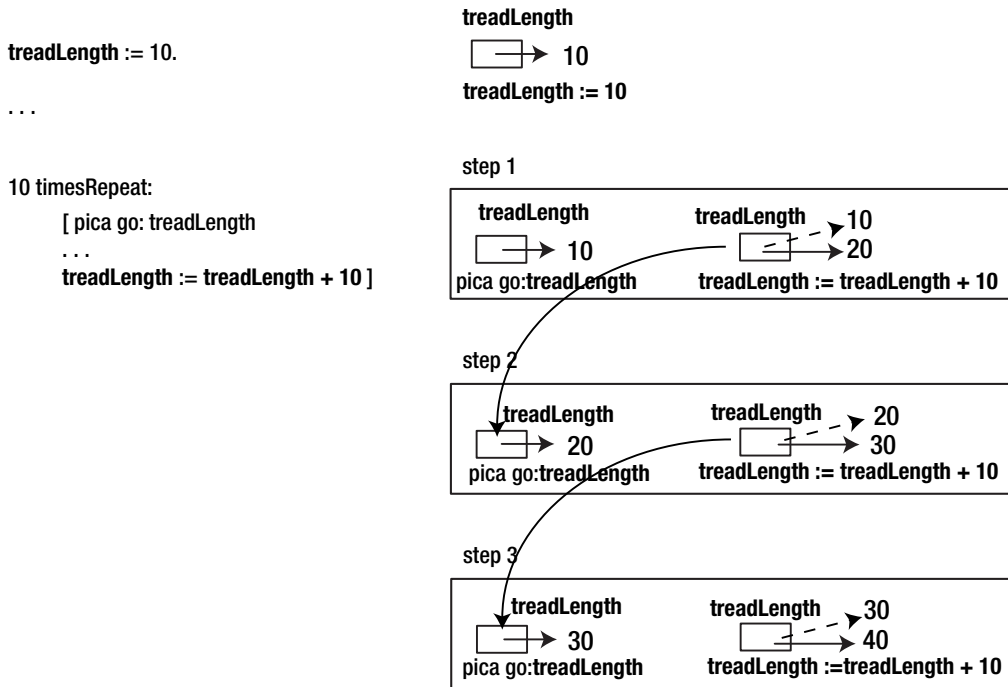
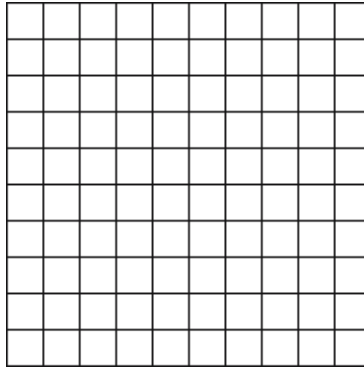


Figure 10-4. The length of a tread is the length of the previous tread plus 10 pixels. The last value that a variable has in the loop body is the value that the variable will have at the start of the loop when it is repeated.

Advanced Experiments

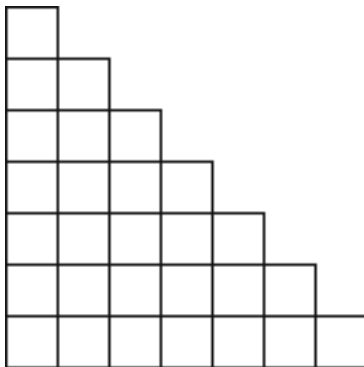
Experiment 10-6 (Squares)

Define a script that creates the checkerboard construction shown below. This experiment is a bit more complicated than the earlier experiments, in that it is difficult to see how to draw the figure using a single loop. However, there are several ways of solving the problem using two loops. For example, you could use one loop to draw all the horizontal lines, and then another loop to draw all the vertical lines.



Experiment 10-7 (Pyramid)

Write a script that creates the construction shown in the figure below, representing the blocks of stones forming part of the step pyramid of Saqqara, which you have seen in previous chapters. You might modify your script from Experiment 10-7 by changing the variable for the length of the lines drawn each time through the loop.



Summary

- When introducing a variable inside a loop, be careful how you initialize the variable. Keep in mind that a variable must be initialized before its value is used. Normally, initialization occurs outside the loop, for otherwise, the value of the variable would not change when the loop is repeated.
- Keep in mind that the last value that a variable is assigned in a loop body will be the variable's value the next time the loop is executed.



Composing Messages

As with any language, Smalltalk follows certain rules in executing the messages sent to objects. I have not yet presented these rules to you, and you may have wondered just what those rules are when you were experimenting with the previous scripts. Your patience will be now rewarded. This chapter explains how to read and write correctly formulated messages.

This chapter may appear a bit more difficult or abstract than the previous ones. However, I have done my best to present clearly the simple rules that govern the writing of messages. Understanding such details might not be as much fun as playing with robots, but it is the price that must be paid if you are to be able to write more advanced programs. The good news is that Smalltalk is not a complex language: there are only five rules that you need to understand. If you are still hesitant, you may also skip this chapter on a first reading and return to it when you have questions about the structure of your programs.

As described in Chapter 2, a message is composed of the *message selector* and the optional *message arguments*. A message is sent to a *message receiver*. The combination of a message and its receiver is called a *message send*.

When you write a complex *expression* such as `pica go: 100 + 20`, it contains two message sends, using the message selectors `go:` and `+`, and you have to know the order in which the messages are executed if you are to understand what the result of the entire expression will be. In Smalltalk, the order in which messages are executed is determined by the *type* of message send. There are three types of messages: *unary*, *binary*, and *keyword-based*. Unary messages are always sent first, followed by binary messages, and finally keyword-based messages. Any messages enclosed in parentheses are executed prior to any other messages. This means that you can change the order in which message sends are executed through the use of parentheses. These rules go a long way toward making Smalltalk code easy to read. And you will soon discover that most of the time, you do not even have to think about them. However, you have to know them, because occasions will arise that will require your knowledge of them.

All of the examples presented in this chapter consist of executable code, as shown in the text. So do not hesitate to try them out and see how they function. I will begin by showing you how to identify the different types of messages, and then I will present some examples of each type. Finally, I will present the rules for message composition.

The Three Types of Messages

Smalltalk defines a small number of simple rules to determine the order in which message sends are executed. These rules, which I will present in detail later, are based on the distinction among three different types of messages:

- *Unary messages* are messages with no arguments. They are sent to an object (the message receiver) without any other information. For example, in the expression `pica color`, the message `color` is a unary message. It does not send any additional information; that is, there is no argument. These messages are called “unary,” from the Latin *unum*, meaning “one,” because a message send with a unary message involves *only one object*, the message receiver.
- *Binary messages* are messages that involve *two objects*: the message receiver and the message selector’s sole argument. (The word “binary” is related to the Latin *bis*, meaning “twice.”) Binary messages are mainly related to mathematical expressions. For example, in the message send `10 + 20`, the message consists of the message selector `+` together with the single argument `20`. The message `+ 20` is sent to the object `10`, which is the message receiver.
- *Keyword-based messages* are messages whose message selector contains a keyword with at least one colon character `:` in its name and that has one or more arguments. For example, in the message send `pica go: 100`, the keyword `go:` contains the colon character `:` and there is one argument, `100`, and therefore there are two objects involved in the message send: the message receiver `pica` and the argument `100`.

Note Don’t be confused by the nomenclature for unary and binary messages. The idea of “one” in the word *unary* and the idea of “two” in the word *binary* refer to the number of objects involved in a message send, not the number of arguments. Thus a unary message has no arguments, and so a unary message send involves one object, namely, the message receiver. A binary message has a single argument, and therefore a binary message send involves two objects: the argument of the message selector and the message receiver.

Identifying Messages

In order to understand the structure of an expression, the first thing you need to do is to identify the messages and their receivers. To do this, I suggest that you use a graphical notation as shown in Figure 11-1. In all the figures of this chapter, the message receivers are *underlined*, each message is surrounded by an *ellipse*, and the messages that make up the expression are *numbered* in the order in which they will be executed. When there is more than one ellipse, the first ellipse to be executed is drawn with a dashed line so that you can see at once where to begin.

Figure 11-1 shows that the expression `pica color: Color yellow` contains within it the expression `Color yellow`. Therefore, there are two ellipses, one for the entire expression `pica color: Color yellow`, and one for the subexpression `Color yellow`. You will learn a bit later that the expression `Color yellow` is executed first, so its ellipse is a broken line and is numbered 1.

Note that each message has a receiver. The robot `pica` receives the message `color: ...`, and `Color` receives the message `yellow`. Therefore, these two message receivers are underlined. (The three dots in the message `color: ...` indicate the argument of the message selector `color:`, which will be the result of the message `send Color yellow`.)

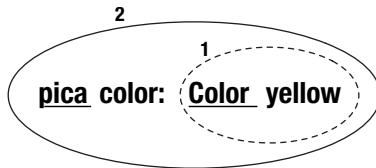


Figure 11-1. *The expression `pica color: Color yellow` contains the subexpression `Color yellow`. The message `yellow` is sent to `Color`, and then the message `color: ...` is sent to `pica`.*

As I have mentioned, every message is sent to an object called the message receiver. A receiver does not have to be a robot. It can be just about anything, from a number to a window. A message receiver can appear explicitly as the first element of an expression, such as `pica` in the expression `pica go: 100` or `Color` in the expression `Color new`. However, a receiver can also be the result of other message sends. For example, in the message `Bot new go: 100`, the receiver of the message `go: 100` is the resulting object returned by the message `send Bot new`. Nevertheless, in every case, a message is sent to some object, and that object is called the message receiver.

Important! In a *message send*, a *message* is always sent to an *object*, called the *message receiver*, which may be named explicitly or may be the result of other message sends.

Table 11-1 shows some message sends, and I suggest that in each case, you identify the type of message and then draw the graphical representation of the expression.

Table 11-1. *Some Examples of Message Sends, Simple and Compound*

Expression	Type(s)	Action
<code>pica go: 100</code>	keyword-based	The receiving robot moves forward 100 pixels.
<code>100 + 20</code>	binary	The number 100 receives the message <code>+</code> with the number 20 as argument.
<code>pica east</code>	unary	The receiving robot <code>pica</code> points to the east.
<code>pica color: Color yellow</code>	keyword-based, unary	The receiving robot <code>pica</code> changes its color to the color <code>yellow</code> .
<code>pica go: 100 + 20</code>	keyword-based, binary	The receiving robot moves forward 120 pixels.
<code>Bot new go: 100</code>	unary, keyword-based	The message <code>new</code> is sent to the <code>Bot</code> class, which returns a new robot to which the message <code>go: 100</code> is sent, causing the robot to move forward 100 pixels.

From Table 11-1 you should observe the following points:

- Some of the messages have arguments, while others do not. The message `east`, being unary, does not have an argument, while `go: 100` and `+ 20` each have one argument, the number 100 and the number 20, respectively.
- Different messages are sent to different objects. In the expression `pica east`, the message `east` is sent to a robot, and in `100 + 20`, the message `+ 20` is sent to the number 100.
- There are simple messages and compound messages. For example, `Color yellow` and `100 + 20` are simple: One message is sent to one object. On the other hand, the expression `pica go: 100 + 20` contains two messages: `+ 20` is sent to 100, and then `go: ...` is sent to `pica`, where `...` represents the result of the execution of the message `send 100 + 20`.
- A message receiver can be the result of an expression that returns an object. In the expression `Bot new go: 100`, the message `go: 100` is sent to the object (a robot) that results from evaluation of the expression `Bot new`.

The Three Types of Messages in Detail

Now that you are able to identify message receivers and the three types of messages, let us look at the different types of messages in detail.

Unary Messages

Among the uses of unary messages are obtaining a value from an object, such as the size of a robot's pen (`pica penSize`); obtaining an object from a class (`Color yellow`); and instructing the receiver to perform an action (`pica beInvisible`). Recall that a unary message does not take an argument: it is sent to the receiver without any other information. Thus the one object involved in a unary message `send` is the message receiver. Unary message sends are of the form of *receiver messageName*. Script 11-1 presents some examples of unary messages, shown in boldface type.

Script 11-1. Examples of unary messages

```
| pica |
pica := Bot new.
pica color.
pica penSize.
pica east.
Color yellow.
125 factorial
```

Important! Unary messages are messages that do not take an argument. A unary message `send` has the form *receiver messageName*.

Binary Messages

Binary messages are messages that involve two objects: the receiver and a single argument. All binary message selectors are composed of one or two characters from the following list: +, *, /, |, &, =, >, <, ~, @. Therefore +, =, and * are message selectors, but so is =>, which is composed of two symbols.

Table 11-2 shows some examples of binary message sends and their meaning. At this point, I would rather not go into the details of these examples, so don't worry if you are not sure about exactly what each of these message selectors does. But try executing the expressions and others like them.

Table 11-2. *Examples of Binary Messages with Numbers*

Expression	Returned Value	Action
1 + 2.5	3.5	Addition of two numbers
3.4 * 5	17.0	Multiplication of two numbers
8 / 2	4	Division of two numbers
10 - 8.3	1.7	Subtraction of two numbers
12 = 11	false	Testing for equality between two numbers
12 ~= 11	true	Testing for inequality between two numbers
12 > 9	true	Is the receiver greater than the argument?
12 >= 10	true	Is the receiver greater than or equal to the argument?
12 < 10	false	Is the receiver less than the argument?
100@10	100@10	Create a point with coordinates (100, 10)

Important! Binary message sends involve two objects: the receiver and a single argument. The message selector of a binary argument is composed of one or two characters from the following list: +, *, /, |, &, =, >, <, ~, @. Binary message sends have the form *receiver messageName argument*.

Keyword-Based Messages

Keyword-based messages are messages that take at least one argument and that contain at least one colon character :. Note that the colon is part of the message selector. Therefore, go:, not go, is the name of a keyword-based message. Script 11-2 shows some examples of keyword-based messages, shown in boldface type.

Script 11-2. *Examples of keyword-based messages*

```
| pica |
pica := Bot new.
pica go: 100.
pica penSize: 5.
pica color: Color yellow.
pica turn: 90
```

I have said that a keyword-based message has at least one argument, but we have not yet seen an example of such a message with multiple arguments. Let us look at an example now: The message `send aNumber between: lowerBound and: upperBound` checks whether the number `aNumber` is in the interval represented by the two numbers `lowerBound` and `upperBound`. This message needs two arguments, namely, the two bounds of the interval. An example is shown in Table 11-3. Note that the message selector is actually `between:and:.` It is composed of the two words `between:and:` and `and:.`

Table 11-3. *Keyword-Based Messages That Take More Than One Argument*

Expression	Arguments	Return Value	Action
<code>5 between: 2 and: 10</code>	<code>2, 10</code>	<code>true</code>	Is 5 between 2 and 10?
<code>Color r: 0 g: 1 b: 0</code>	<code>0, 1, 0</code>	a green color object	creates a color with the given values of red, green, and blue.

Important! Keyword-based messages have at least one argument, and their message selector contains at least one colon character `:`. A keyword-based message `send` that takes two arguments is of the form `receiver messageNameWordOne: argumentOne messageNameWordTwo: argumentTwo`.

Order of Execution

You have seen that there are three kinds of messages: unary, binary, and keyword-based. Now I will tell you, as I promised, how to determine the order in which messages are executed. The order of message execution is determined by the type of message, as described by the following three rules:

Rule 1: Unary message sends are executed first, then binary message sends, and finally keyword-based ones.

Rule 2: As with mathematical expressions, the priority of message execution can be overridden by parentheses: message sends in parentheses are executed before any other types of message sends.

Rule 3: Message sends of the same type are executed from left to right.

These rules may seem complex, but they are quite natural, and once you get used to them, you will not have to think too much about them most of the time. In particular, the third rule simply states that messages of the same type are executed in the order in which they are read.

If you are ever in doubt and want to be sure that your messages are executed the way you want, you can always add extra parentheses, as shown in Figure 11-2. In the figure, the expression `pica color: Color yellow` is analyzed. The message selector `yellow` is a unary message, while the message selector `color:` is a keyword-based one. Therefore, the expression `Color yellow` is executed first. If you are unsure about the order of execution, then you can put

parentheses around `Color yellow` to make sure that it will be executed first. This won't change the natural order of execution. That is, `pica color: Color yellow` and `pica color: (Color yellow)` have precisely the same effect. The rest of this section illustrates each of these points.



Figure 11-2. *Unary message sends are executed first, so `Color yellow` is executed first. This execution returns a color object, which is passed as the argument of the message `color: ...` that is sent to `pica`.*

Rule 1: Unary > Binary > Keywords

Unary message sends are executed first, then binary message sends, and finally keyword-based message sends. In programmer jargon we also say that unary messages have *precedence* over binary messages, and binary messages have precedence over keyword-based messages.

Important! Rule 1: Unary message sends are executed before binary message sends, which are executed before keyword-based message sends.

Example 1

In the message send `pica color: Color yellow`, there is one unary message, `yellow`, sent to the class `Color`, and there is one keyword-based message, `color: ...`, which is sent to the robot `pica`. Unary message sends are executed first, so the first message send to be executed is `Color yellow`. This execution returns a color object, represented as `aColor` (since we need a name to refer to it), which is passed to `pica` as the argument of the message `color: aColor`. Figure 11-2 shows graphically the order in which the messages are executed.

As an aid to your understanding, I would like to propose a textual way of representing a compound message send in step-by-step execution. In Step-by-step 11-1, the message send to be executed step by step is `pica color: Color yellow`. The first line shows the complete message send in boldface type.

Step-by-step 11-1. *Decomposition of the execution of `pica color: Color yellow`*

```

pica color: Color yellow
(1)      Color yellow      "unary"
         -returns> aColor
(2) pica color: aColor      "keyword-based"
```

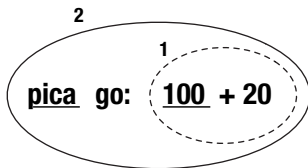

The lines of code represent the execution steps in numbered order in which they will occur. Thus `Color yellow` is the first expression to be executed. Note that the expressions have been indented to line up with their counterparts in the message send at the top.

When the execution of a message send returns a result that is used in the following execution, the line following the executed expression shows “*-returns>*” followed by the result. Here the expression `Color yellow` returns a color object that I have called *aColor* so that it can be referred to in the sequel. The second expression to be executed is `pica color: aColor`, where as I just explained, *aColor* is the result obtained from the previous execution step. To stress this point, the returned value or object is displayed in italic type. For further clarification, the kind of message that is currently being executed is displayed as a comment inside quotation marks. For example, `Color yellow` is shown to be a unary message.

Example 2

The message send `pica go: 100 + 20`, contains the binary message selector `+` and the keyword-based message selector `go:`. Binary messages are executed prior to keyword-based messages, so `100 + 20` is executed first: The message `+ 20` is sent to the object 100, which returns the number 120. Then the message `pica go: 120` is executed with 120 as argument. Step-by-step 11-2 shows how the expression is executed.

Step-by-step 11-2. *Decomposition of the expression* `pica go: 100 + 20`



```

pica go: 100 + 20
(1)      100 + 20      "binary"
          -returns> 120
(2) pica go: 120      "keyword-based"

```

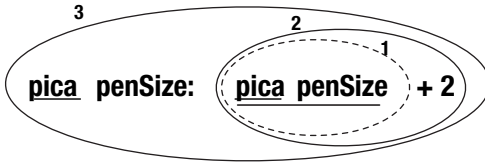
Example 3

The message `pica penSize: pica penSize + 2` contains the unary message `penSize`, the binary message with selector `+`, and the keyword-based message with selector `penSize:`. Step-by-step 11-3 illustrates the decomposition of message execution. The unary message send `pica penSize` is executed first (step 1). This message returns a number, which we are calling *aNumber*, representing the current size of the receiver’s pen. Then the binary message send `aNumber + 2` is executed (step 2). The number *aNumber* is the receiver of the message `+ 2`, which in turn returns another number, the sum, which here is called *anotherNumber*. Finally, the keyword-based message `penSize: anotherNumber` is sent to `pica`, who sets his pen size to *anotherNumber*.

Altogether, the entire compound expression increases the receiver’s pen size by two pixels. It does so by first asking `pica` for his pen size (`pica penSize`), increasing that number

by two ($aNumber + 2$), and then telling `pica` to change his pen size to the new number (`pica penSize: anotherNumber`). Note that `penSize` and `penSize:` are two different message selectors! The first is unary and *asks* the receiver for its pen size, and the second is keyword-based and *tells* the receiver to change its pen size to the value of its argument.

Step-by-step 11-3. *Decomposition of the expression* `pica penSize: pica penSize + 2`



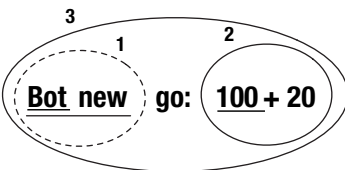
- ```

pica penSize: pica penSize + 2
(1) pica penSize "unary"
 -returns> aNumber
(2) aNumber + 2 "binary"
 -returns> anotherNumber
(3) pica penSize: anotherNumber "keyword-based"

```

#### Example 4

As an exercise, I will let you decompose the execution of the message `Bot new go: 100 + 20`, which is composed of one unary, one keyword-based, and one binary message (see Figure 11-3).



**Figure 11-3.** *Decomposing* `Bot new go: 100 + 20`

## Rule 2: Parentheses First

The default ordering of message execution may not be suitable for what you want to accomplish in an expression, and so you should be able to change it. For this purpose, Smalltalk offers parentheses ( and ). Just as in mathematics, expressions in parentheses get the highest precedence, and they are executed before any others.

Keep in mind that if you find the rules for order of execution a bit complex or if you simply want to clarify the structure of an expression, use parentheses to ensure that the messages are executed in the order that you wish. Figure 11-4 shows some of the expressions that we have previously looked at together with their equivalents using parentheses.



## Rule 3: From Left to Right

Now that you know how messages are categorized according to priority of execution, the final question to be addressed is how messages with the same priority are executed. Rule 3 states that they are executed from left to right. This rule was used already, in Step-by-step 11-4, where the left-hand binary message @ 325 was executed before the right-hand message @ 100.

---

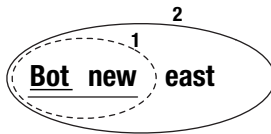
**Important!** Rule 3: Messages of the same type are executed in order from left to right.

---

### Example 6

In the expression `Bot new east`, both message sends are unary messages, so the first one as you read from left to right, `Bot new`, is executed first. It returns a newly created robot, called `aBot` in Step-by-step 11-5, to which the second message, `east`, is sent. Step-by-step 11-5 shows the order of execution.

**Step-by-step 11-5.** *Decomposition of the expression `Bot new east`*

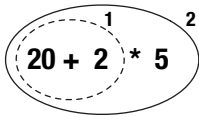


**Bot new east**

- (1) `Bot new` "unary"  
-returns> `aBot`
- (2) `aBot east` "unary"

### Example 7

In the expression `20 + 2 * 5`, there are only the two binary message selectors `+` and `*`. According to Rule 3, since `+` is to the left of `*`, it should be executed first. In normal mathematical notation as well as in many programming languages, multiplication would take precedence over addition regardless of the order in which the arithmetic operations appear. However, in Smalltalk there is *no specific priority* for mathematical operations. The message descriptors `+` and `*` are just binary messages, and therefore they have equal status. The message selector `*` does not have precedence over `+`, and the leftmost message selector `+` is sent first, and then `*` is sent to the result, as shown in Step-by-step 11-6.

**Step-by-step 11-6.** *Decomposition of  $20 + 2 * 5$* 

```

20 + 2 * 5
(1) 20 + 2
 -returns> 22
(2) 22 * 5
 -returns> 110

```

---

**Note** There is no priority among binary messages. In the expression  $20 + 2 * 5$ , the leftmost message `+` is evaluated first, despite the fact that in normal mathematical notation, multiplication takes precedence over addition.

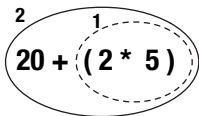
---

You can see, then, as shown in Step-by-step 11-6, that the result of this expression is not 30, which you would get if you did the multiplication first, but 110. This behavior is surprising at first, but it derives from the three simple rules for executing messages. This counterintuitive order of mathematical operations is the price that we have to pay for the simplicity of the Smalltalk model, which has only methods. If you want your expression to obey the normal priority of mathematical operations, then you should use parentheses, since when message sends are enclosed in parentheses, they are executed first. Hence the expression  $20 + (2 * 5)$  returns the result 30, as shown in Step-by-step 11-7.

---

**Important!** Message sends surrounded by parentheses are executed first. Therefore, in the expression  $20 + (2 * 5)$ , the message with `*` is executed before the one with `+`, which is the usual order of operations in mathematics.

---

**Step-by-step 11-7.** *Decomposition of  $20 + (2 * 5)$* 

```

20 + (2 * 5)
(1) 2 * 5
 -returns> 10
(2) 20 + 10
 -returns> 30

```

---

**Note** In Smalltalk, the mathematical message selectors such as + and \* all have the same priority. The symbols + and \* are simply message selectors for binary messages. Therefore, \* does not have priority over +. If you want to force one operation to take precedence over another, then you should use parentheses.

The fact that Smalltalk does not follow mathematical precedence can be confusing at the beginning. Therefore, when you have multiple binary messages representing a mathematical expression, give yourself and anyone else reading your program a break and insert parentheses to express how the computation should be performed. When you have become more accustomed to the way that messages are executed, you will probably become more *laissez-faire* about parentheses.

---

A consequence of rule 1—which provides for the order of execution of different types of messages, with unary messages being executed before binary messages, and binary messages before keyword-based messages—is that you very often do not have to use parentheses. That is, most of the time, you do not have to worry about order of execution. Table 11-4 shows expressions written to be executed according to Smalltalk's rules and equivalent expressions using parentheses, which would be necessary if the order of precedence did not exist.

**Table 11-4.** *Some Expressions and Their Fully Parenthesized Equivalents*

| Without Parentheses            | Equivalent Expression with Parentheses |
|--------------------------------|----------------------------------------|
| pica color: Color yellow       | pica color: (Color yellow)             |
| pica go: 100 + 20              | pica go: (100 + 20)                    |
| pica penSize: pica penSize + 2 | pica penSize: ((pica penSize) + 2)     |
| 2 factorial + 4                | (2 factorial) + 4                      |

## Summary

- A message is always sent to an object, called the message receiver, which may be the result of other messages.
- Unary messages are messages that take no argument. A unary message send is of the form of *receiver messageName*.
- Binary messages are messages that involve two objects: the receiver of the message and a single argument. The message selector of a binary message consists of one or two characters from the following list: +, \*, /, |, &, =, >, <, ~, @. Binary message sends are of the form *receiver messageName argument*.
- Keyword-based messages are messages that take one or more arguments and use a keyword with at least one colon character :. A keyword-based message send taking two arguments is of the form *receiver messageNameWordOne: argumentOne messageNameWordTwo: argumentTwo*.

- Rule 1. Unary messages are executed first, then binary messages, and finally keyword-based messages.
- Rule 2. As in mathematics, expressions in parentheses are executed before any others.
- Rule 3. When messages are of the same type, the order of execution is from left to right.
- In Smalltalk, mathematical message selectors such as + and \* have the same priority, and therefore \* does not have priority over +. You should use parentheses to ensure that your mathematical expressions are executed in the proper order.

## PART 3



# Bringing Abstraction into Play

**I**n this part of the book you will learn how to define your own methods. This will enable you to reuse sequences of messages, and you will be able to define complex methods out of simpler ones.





# Methods: Named Message Sequences

**U**p to now, you have been using scripts to create robots and send them sequences of messages. Using scripts has the advantage of being a straightforward approach, but it has some severe limitations. One of the major limitations is that a script cannot be called by another script. This is a serious problem, because a script cannot be reused by other scripts. You have to rewrite the same sequence of messages again and again.

Wouldn't it be nice if one could define a kind of script whose sequence of messages could be sent to any robot? In fact, this is possible, and such a sequence of messages is called a *method*. (In the context of this book we will not go into the full power of methods, since that would get us into the somewhat tricky subject of object-oriented programming.) A *method* is a named script. The name of a method can be used in a script or even in another method to invoke the method. Actually, there is nothing much new here: all the robot messages that you have used so far represent methods that you could use with any robot!

In this chapter, you will learn how to define methods. You already know most of what you need to write the code of a method. However, a method must be defined using a special editor called a *code browser*. We will start by comparing a script and a method. Then we will define a method, and finally, we will look in detail at what we have accomplished.

## Scripts versus Methods

Let's look at one of the scripts that you have already written, for example, Script 12-1, which creates a robot and tells it to draw a square with side length 100 pixels.

**Script 12-1.** *Pica draws a simple square.*

```
| pica |
pica := Bot new.
4 timesRepeat:
 [pica turnLeft: 90.
 pica go: 100]
```

The problem with this script is that each time you need to draw a square of side length 100 you need to *copy* the three last lines of Script 12-1. Furthermore, if you want another robot (for example, daly) to draw the square, you must change the name *pica* to *daly* everywhere. This is illustrated by Script 12-2.

**Script 12-2.** *Pica and daly each draw a simple square.*

```
| pica daly |
pica := Bot new.
daly := Bot new.
daly jump: 200.
daly color: Color red.

4 timesRepeat:
 [pica turnLeft: 90.
 pica go: 100].
4 timesRepeat:
 [daly turnLeft: 90.
 daly go: 100].
```

For all these reasons, working with scripts is not easy. In fact, I suspect that the following three statements reflect your personal experience with scripts:

- Writing long scripts is a painful task.
- Repeating long scripts is boring and error-prone.
- When one is copying complex scripts, the likelihood of making a programming error, such as omitting a line, is high. (A programming error is an error in the logic of a program. In contrast to syntax errors, which are caught quickly by the computer because they are errors in program structure, programming errors can be quite difficult to catch.)

To overcome these difficulties, we would like to *define* a sequence of messages once and for all, give the sequence a *name*, and then be able to *send the named sequence* as a single message to any robot, just as we have been able to send predefined robot messages such as `go:`, `north`, and `jump:`.

With this approach we could define a new *method* called `square`, and then write Script 12-3. But don't execute the script yet, because the method `square` has not yet been defined. Once you have the method `square`, you will no longer have to copy and adapt the sequence of messages defining a square. You can simply use it twice. The message `send pica square` will tell `pica` to carry out the instructions encoded in the method `square`.

I hope that I have convinced you that defining methods will be worth the effort.

**Script 12-3.** *Pica and daly draw squares using the method square.*

```
| pica daly |
pica := Bot new.
daly := Bot new.
daly go: 200.
daly color: Color red.
pica square.
daly square
```

## How Do We Define a Method?

In this section I will give you a cookbook recipe for creating a method. In Squeak you can define methods on any object, but in this book you will define methods only for robots. To help you out with this, I developed a specialized code browser named Class Bot Browser just for defining methods for your robots. There is a Class Bot Browser in the working flap, or you can always create one by dragging its thumbnail from the dark blue flap or via the menu **open....**

Using a Class Bot Browser to define a method requires you to (1) choose or create a method category, which is a kind of method folder; (2) type the method; and then (3) compile it. These steps will be described in upcoming sections. But first let us have a detailed look at the different parts of a Class Bot Browser.

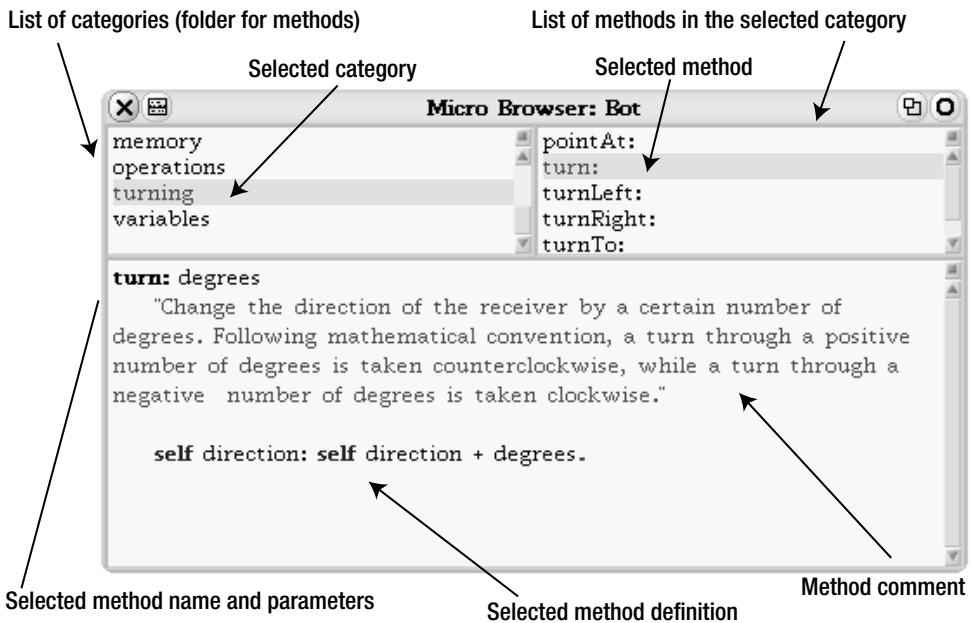
## A Class Bot Browser

Defining methods requires a new tool: the editor shown in Figure 12-1. This browser is actually a simplified version of the browser used by Smalltalk programmers. The browser consists of three parts, or *panes*:

**Categories.** The upper left pane contains the *category list*. It shows the different method categories. Method categories are just names that group methods together so that you can find information faster. In Figure 12-1, the category *turning* is selected; it groups all the operations having to do with robots' directional changes. Other categories that group other robot methods are also listed.

**Methods.** The upper right pane contains the *method list*. This list shows the method names of the methods in the selected category. In Figure 12-1, five methods are listed: *pointAt:*, *turn:*, *turnLeft:*, *turnRight:*, and *turnTo:*. The method named *turn:* is currently selected.

**Method Definition.** The bottom pane contains the *code editor*. It shows the definition of the method whose name is selected together with optional comment text. This pane is also the place where you can type the code of a new method.

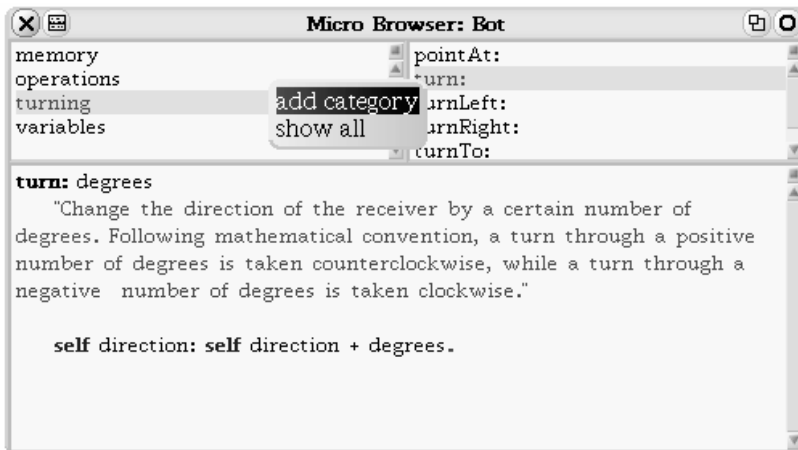


**Figure 12-1.** A Class Bot Browser showing the definition (bottom pane) of the method *turn:* (selected in the upper right pane) belonging to the category *turning* (selected in the upper left pane).

## Creating a New Method Category

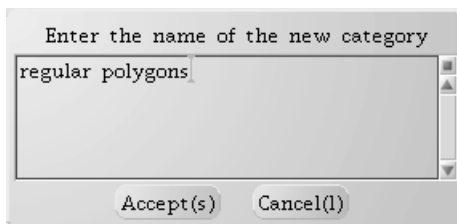
Methods are grouped by categories. A category is defined by giving it a name. To define a method, you either define a new category for it or select an existing category. Let's create a new category named `regular_polygons`. Here is how it is done:

1. Click with the right mouse button (Alt-click or Option-click) on the category list. A menu like the one in Figure 12-2 will pop up.



**Figure 12-2.** To create a new method category, open the category menu and select `add category`.

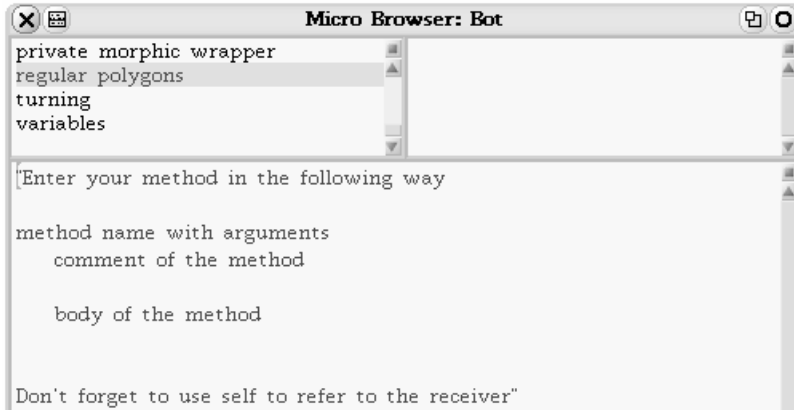
2. Select the option `add category` of that menu.
3. Type the name of the category in the dialog box that appears, as shown in Figure 12-3. You may choose any name for the category. Of course, meaningful names are better than meaningless ones when you want to share your work with other people or find your method again at a later date.



**Figure 12-3.** Enter a new category name in the dialog box and click the `Accept` button.

4. Click the `Accept` button to validate your choice.

As shown in Figure 12-4, the name of the new category appears in the category pane and is automatically selected. The editor is ready to accept a new method definition. It shows you a reminder of how to define a method, which you can remove when you start typing your method. You are now ready to define your first method.



**Figure 12-4.** *The new category is ready.*

## Defining Your First Method

If the category to which you want to add your method is not selected, select it. Then type the contents of Method 12-1 (following this paragraph) into the code editor pane. To do this, select all the text in the code editor and start typing your method.

**Method 12-1.** *A new method for drawing a square of side length 100*

**square**

*"Draw a square of side length 100 pixels"*

```

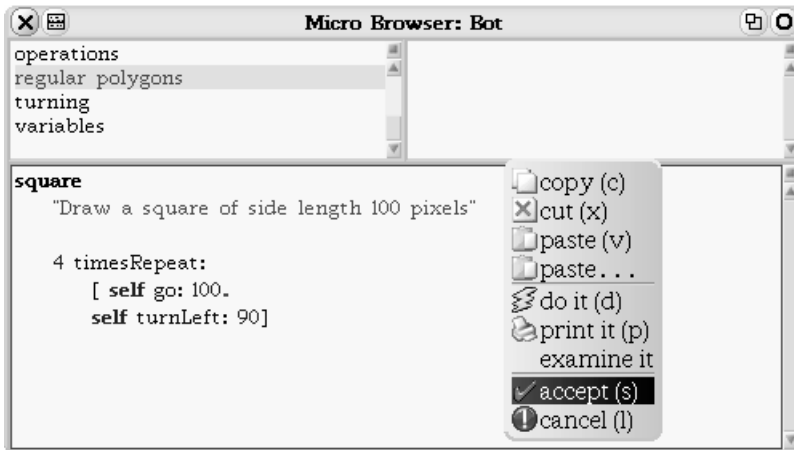
4 timesRepeat:
 [self go: 100.
 self turnLeft: 90]

```

Defining a method is a three-step process:

1. **Typing the method.** Typing code into the code editor pane works exactly as with the script editor. First delete the reminder text that is in the code editor pane. The easiest way to do this is to point your mouse at the beginning of the editor pane before the first character and click. This will select all of the code editor text. Once you finish typing the new method, your code display pane should look like Figure 12-5.

2. **Compiling the method.** Click to bring up the menu for the code editor, as shown in the figure, and select the option **accept**. Doing so causes the method definition to be *compiled*, that is, transformed into a representation that the computer can understand and execute. A new method named `square` now appears in the method list. If you made a mistake while typing the method, Squeak will report the error as it would for a script.



**Figure 12-5.** After typing in the method `square`, you compile it using the code editor menu.

If you defined the method correctly, you should be able to compile it without Squeak reporting any errors. The browser will then reflect the fact that the compilation is complete and that robots can now understand messages containing the new method by showing the new method's name in the method list (see Figure 12-6).

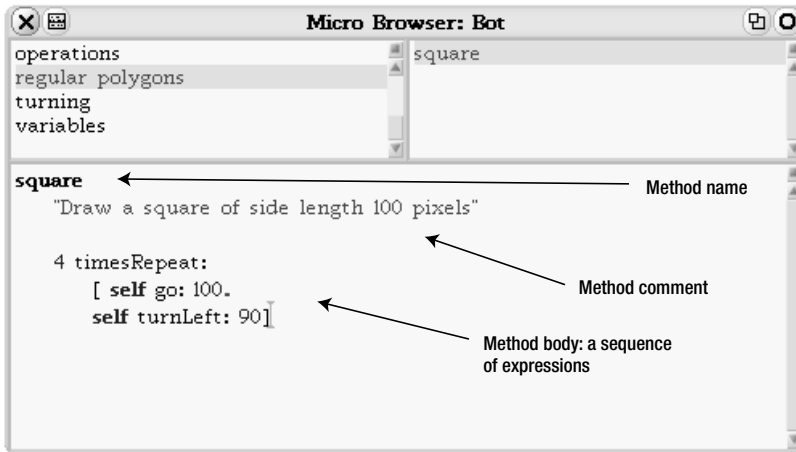
3. **Testing the method.** As the saying goes, the test of whether a pudding has been properly made is in tasting it. Likewise, you have not finished creating your new method until you have tested it, because the method that you defined might not do what you had in mind for it. Now you may execute Script 12-3. You should get one black square and one red square.

Observe that a method can be used and reused, as demonstrated by Script 12-3. This is old news. Indeed, you have used this fact since the beginning of this book: message selectors such as `go:` and `turnLeft:` are the names of methods defined in the same way as the method `square`.

## What's in a Method?

I asked you to type a method without much explanation. Now it is time to analyze the structure of the method.

A method is composed of a *name*, an optional *method comment*, and a *method body* (a sequence of expressions), as shown in Figure 12-6. The method name can also contain parameters (see Chapter 14), and the method body can also define local variables using vertical bars `|`.



**Figure 12-6.** A method is composed of a name, an optional method comment, and a method body.

**Method name.** A method name should always represent what the method does, not how it does it. When you want somebody to open a door, you don't explain all the physics and mathematics involved. It is the same for methods.

---

**Important!** A method name should always represent what the method does, not how it does it.

---

Method names without parameters, such as `square`, follow the same syntax as variable names. They are composed of alphanumeric characters (letters and digits) and start with a lowercase character. In our case, the method name is `square`.

**Method comment.** A comment consists of text enclosed between double quotes ("This is a comment"). The text itself cannot contain any double quotes. However, a comment can be as long as you like, and can continue over several lines.

In general, a comment explains the purpose and the effect of the method. It explains how the method can be used, but not how the method does its job. Anyone who wants to know how the method works can read the method's body.

If the method name is clear enough, the comment may be omitted. In our case the method comment is, "Draw a square of side length 100 pixels".



**Method body.** After the comment comes the method definition itself, which is the sequence of messages that are executed in response to a message. In our case, the method body is as follows:

```
4 timesRepeat:
 [self go: 100.
 self turnLeft: 90]
```

---

**Important!** A method is a named sequence of expressions. It is composed of a name, an optional comment, and a sequence of expressions. Once a method for robots has been defined, any robot can execute it in response to a message with the same name.

---

## Scripts versus Methods: An Analysis

Let's compare Method 12-2 with Script 12-4. You can see three significant differences: (1) The line in the script declaring the variable `pica` is not in the method; (2) the line creating the robot is also not in the method; (3) in the remainder of the method, the variable `pica` is replaced by `self`.

**Script 12-4.** *Pica draws a simple square.*

```
| pica |
pica := Bot new.
4 timesRepeat:
 [pica turnLeft: 90.
 pica go: 100]
```

**Method 12-2.** *Instructions to any robot for drawing a simple square.*

```
square
 "Draw a square of side length 100 pixels"
 4 timesRepeat:
 [self go: 100.
 self turnLeft: 90]
```

Remember that a robot method represents a sequence of expressions that can be sent to *any* robot: The robot in the script referred to by the variable `pica` will not necessarily be the receiver of the message `square`. The robot `daly`, or any other robot, could also be the receiver of the message `square`, as we saw in Script 12-4.

Therefore, it is important in defining the method `square` not to refer to any *particular* robot, since the message `square` will be sent to *different* robots at different times. Thus we need a name that will stand for whatever robot happens to be the message receiver of the message `square`. That is the purpose of `self`. Inside a method, `self` represents the object receiving the message, because that object *itself* will be executing messages such as `go:` and `turnLeft:`.

## The Variable “self”

In Chapter 8 I explained that a variable is just a named placeholder for an object. In particular, I emphasized that the same variable could be used to point to different objects at different times.

In the case of a method, the variable `self` points to whatever object has received the message: when the expression `pica square` is executed, the variable `self` in the method `square` refers to the robot named `pica`, and when the expression `daly square` is executed, `self` refers to the robot named `daly`.

---

**Important!** Inside a method, the variable `self` represents the object that has received the message that led to the execution of that method. For example, when the expression `pica square` is executed, `pica` receives the message `square` and executes the robot method of the same name. The word `self` in the method now refers to the robot named `pica`, since `pica` is executing the method; when the expression `daly square` is executed, `self` refers to the robot named `daly`.

---

The word `self` in a method is a special sort of variable, because you cannot change its value. Only Squeak can assign the value of `self`. That is why `self` is not declared between vertical bars `| |`. Moreover, `self` can be used only inside a method definition.

---

**Important!** When the code of a method needs to send a message to the receiver, the message is sent to `self`. For example, in the method `square`, the robot executing the method needs to turn *itself*, so the message `turn: 90` is sent to `self`.

---

## Method or Not: That Is the Question

At this stage, you may be tempted to go back and convert all the scripts you have written into methods. This is not advisable, because not every script is worth turning into a method. In general, you should define a method when you have a sequence of messages that is general enough to be used several times.

## Returning a Value

A method can also return a value by using the character `^`, called a *caret*. When you type a caret, Squeak prints an upward-pointing arrow (`↑`) in the environment. Imagine that you want to have a method that returns the greatest distance that a robot should be allowed to move at one time. You could define the method `maxDistance`, shown in Method 12-3. In this example, the method simply returns a number, but instead, you could have the method return the result of a complex expression, perhaps involving where the robot is positioned on the screen.

**Method 12-3.** *This method returns a value.*

```
maxDistance
 "returns the maximum distance a robot should be able to move"
 ^ 100
```

If a method does not explicitly return a value, then it returns the message receiver by default. Method 12-4 is equivalent to the method `square` defined previously. In fact, at the end of every method there is an implicit expression `^ self` if there is no explicit return expression. However, in this book you do not have to worry about that.

**Method 12-4.** *This equivalent version of the `square` method explicitly returns the message receiver.*

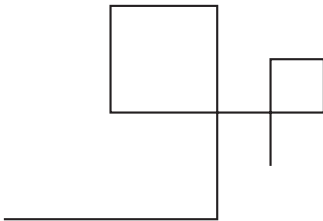
```
squareEquivalent
 "Draw a square of side length 100 pixels"
 4 timesRepeat:
 [self go: 100.
 self turnLeft: 90].
 ^ self
```

In this book, you will not use this feature much, but it is important to know that a method always returns a value.

## Drawing Patterns

Now it is time to practice. As you have seen, it is quite easy to transform a script into a method. Many seasoned programmers use scripts to test ideas. When they have proven the feasibility of an idea in the form of a script, they move the code of the script into a method for later reuse. The next exercise trains you to do exactly this. Let's consider Script 12-5, which draws an abstract "art nouveau" design.

**Script 12-5.** *Pica draws a simple abstract pattern.*



```
| pica |
pica := Bot new.
pica go: 100 ;
 turnLeft: 90 ;
 go: 100 ;
 turnLeft: 90 ;
 go: 50 ;
```

```

turnLeft: 90 ;
go: 50 ;
turnLeft: 90 ;
go: 100 ;
turnLeft: 90 ;
go: 25 ;
turnLeft: 90 ;
go: 25 ;
turnLeft: 90 ;
go: 50

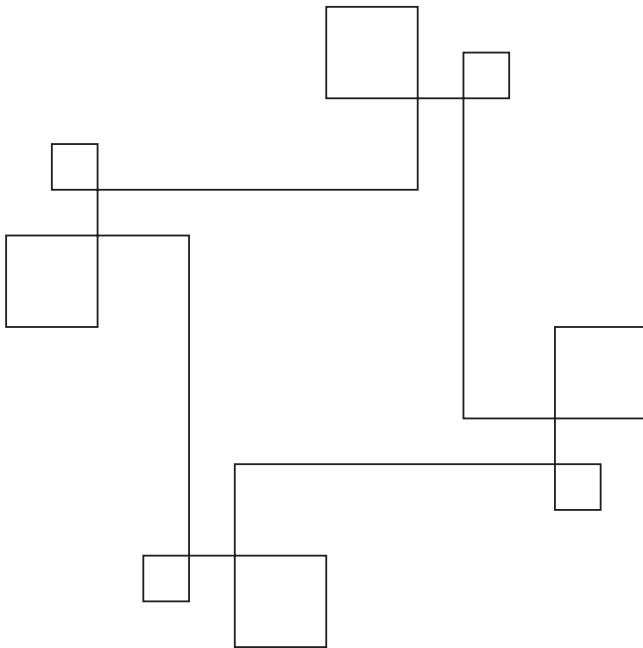
```

### Experiment 12-1 (A Simple Abstract Design)

Create a method named `pattern` that produces the figure drawn by Script 12-5.

You now can use this method in a script to draw a more elaborate design that might be used for an art nouveau picture frame.

**Script 12-6.** *An art nouveau picture frame*



```

| pica |
pica := Bot new.
4 timesRepeat: [pica pattern ; go: 50]

```

At this point, the astute reader might ask, Why don't we create a method, named `frame50`, for example, corresponding to that of Script 12-6? This is indeed possible, since *any method created for a robot can be reused by another robot method*. Creating such methods is the topic of the next chapter.

### Experiment 12-2 (A Method for the Art Nouveau Picture Frame)

Create a method named `frame50` that produces the design produced by Script 12-6.

## Summary

- A *method* is a named sequence of expressions. It is composed of a name, a comment, and a sequence of expressions. Once a method for robots has been defined, any robot can execute it in response to a message with the same name.
- A method name should always represent what the method does, not how it does it.
- A new method for a robot is created using a Class Bot Browser, which is a special editor for defining methods.
- Inside a method, the variable `self` represents the object that receives the message. When the method's code needs to send a message to the receiver, the message should be sent to `self`.

## Glossary

**Method categories.** A method category is a folder in which methods are sorted. Categories help you to find methods more quickly.

**Method.** A method represents a sequence of expressions that an object can execute. A method has a name. It is executed when an object receives a message having the same name.

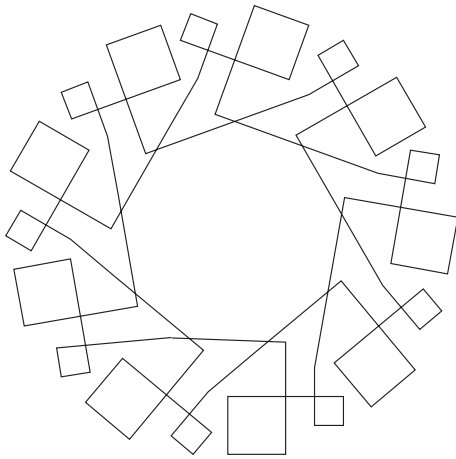
**Class Bot Browser.** A Class Bot Browser is a special tool for viewing and editing methods.

**Comment.** A comment is a piece of text surrounded by quotation marks that explains the purpose of a method.

**self.** The variable `self` is predefined by Smalltalk. It always represents the receiver of the message in a method definition.



# Combining Methods



In Chapter 12, you learned how to define methods. I showed that defining methods is interesting and useful because (1) methods save you from having to rewrite scripts, which is time-consuming and subject to error, and (2) methods can be used and reused by different robots. The other main advantage of using methods is the possibility of using methods in other methods, that is, calling one or more existing methods as part of the definition of a new method. The reuse of methods is what we will explore in this chapter.

Being able to reuse methods is extremely important, because we can define a method in terms of another one without having to know all the details of how the second method is defined. We just call it and ask it to do what it is designed to do.

## Nothing Really New: The Square Method Revisited

Having methods call other methods (which we call *composing methods*) is quite natural and is not really new. In fact, it is what you did in Chapter 12 when you defined a method! The method `square` includes in its definition calls to the methods `turnLeft:`, `go:`, and `timesRepeat:` (as shown in Method 13-1). Thus even the simple method `square` is defined in terms of other methods, and we did not have to know how `turnLeft:`, `go:`, and `timesRepeat:` are defined. We needed to know only what they do. So we are essentially done with this chapter, with nothing left to do but have some fun.

### Method 13-1

#### `square`

```
"Draw a square of 100 pixels wide "
```

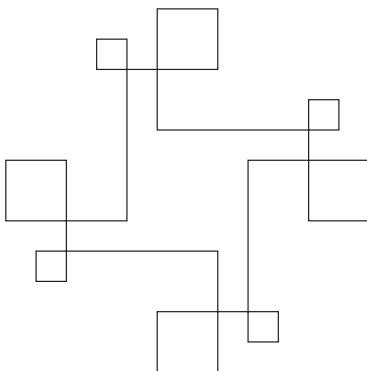
```
4 timesRepeat:
[self go: 100;
 turnLeft: 90]
```

## Other Graphical Patterns

In Chapter 12, I asked you to define the method `pattern`, which draws a simple abstract pattern (See script 12-5). Now I will ask you to perform some further experiments that will produce more drawings by defining more methods.

### Experiment 13-1

Define a method `pattern4` that calls `pattern` four times to produce the figure below. You will use this method later, in another script. After you have created the method `pattern4`, use the following three-line script to make `pica` draw the figure:



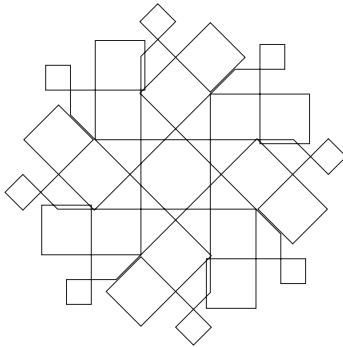
```
| pica |
pica := Bot new.
pica pattern4
```

### Experiment 13-2 (A Ferris Wheel)

Define a method called `tiltedPattern` that draws the picture at the beginning of this chapter, which looks somewhat like a Ferris wheel. Hint: you will have to call `pattern` nine times, and the angle through which to turn between calls is 10 degrees.

### Experiment 13-3 (Doubling the Frame)

Define the method `doubleFrame`, presented below, that draws the picture shown after the method definition.



#### `doubleFrame`

```
8 timesRepeat:
 [self pattern.
 self turnLeft: 45.
 self go: 100]
```

## What Do These Experiments Tell You?

Now let's see what you can learn from the experiments you did. As you can see from the methods `pattern4`, `tiltedPattern`, and `doubleFrame`, the method `pattern` was defined only once, and then *reused several times* in different methods. Defining `pattern` as a method allows you to (1) define it only once, (2) reuse it in various contexts, and (3) not introduce errors by copying this method over and over.

If you look at the definition of the method `doubleFrame`, you see that it is defined in terms of the `pattern` method, which is itself defined in terms of other methods, such as `go:` and `turnLeft:`. In fact, a complex method is often defined in terms of simpler methods, which themselves are defined in terms of even simpler methods, which themselves are defined in terms of even simpler methods, which themselves.... The advantage of this is that it is easier to understand and to define simple methods than complex methods, and the technique of defining methods in terms of simpler methods limits the degree of complexity in any one method. In Chapter 16, I will show you that to solve a problem, it is advantageous to decompose it into smaller subproblems, solve these subproblems, and then use the solutions to the smaller subproblems to solve the main problem.



It is essential to understand that in defining the method `doubleFrame`, you do not have to know how `pattern` is defined. You just need to know what it does and how to use it! When we define a method, we are giving a single name to a sequence of messages, which reduces the number of details that we have to keep track of. We just have to remember what the method does and its name, not how it does it. We say that we are building an *abstraction* over the definition details.

To make this point clear, I rewrote the method `doubleFrame` without calling the method `pattern` by directly copying the definition of `pattern` (shown in italics). Compare `doubleFrameWithoutCallingPattern` (Method 13-2) with the method `doubleFrame`. The new version without `pattern` is not only longer, but for most people it is also more confusing and harder to understand.

Now imagine what would happen if I did the same with the code of `turnRight:`, `turnLeft:`, and `go:`—because these are methods too. It would be a nightmare! There would be so many details that we would be lost all the time.

**Method 13-2.** *Creating the double frame without the abstraction of the pattern method*

**doubleFrameWithoutCallingPattern**

```
8 timesRepeat:
 [self go: 100.
 self turnRight: 90.
 self go: 100.
 self turnRight: 90.
 self go: 50.
 self turnRight: 90.
 self go: 50.
 self turnRight: 90.
 self go: 100.
 self turnRight: 90.
 self go: 25.
 self turnRight: 90.
 self go: 25.
 self turnRight: 90.
 self go: 50.
 self turnLeft: 45.
 self go: 100]
```

---

**Important!** When you write a new method, it can call other methods. You can use a method without knowing how it is written. After you finish writing a method, you can call it when you write another method.

---

# Squares Everywhere

Now it is time to practice. Define the following methods using the method square.

## Experiment 13-4 (Some Boxes)

Define methods `box` and `separatedBox` that produce the pictures shown in Figure 13-1.

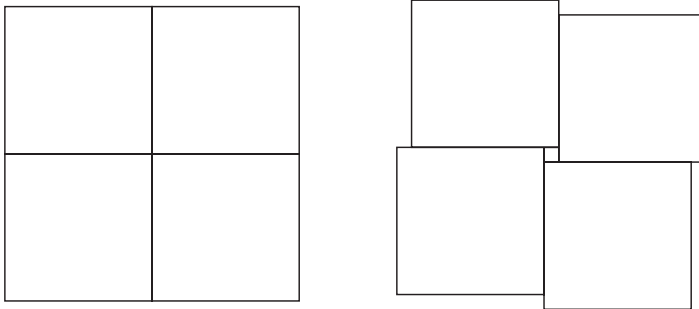


Figure 13-1. *Boxes*

## Experiment 13-5 (Your Choice)

Use your previous methods to generate various figures of your choice. Have fun!

## Experiment 13-6 (A Star)

Using the method `box`, experiment and define a method `star` that produces the right-hand picture in Figure 13-2.

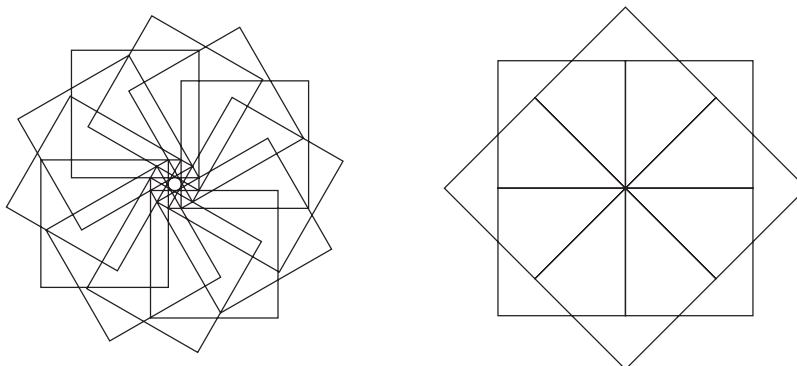


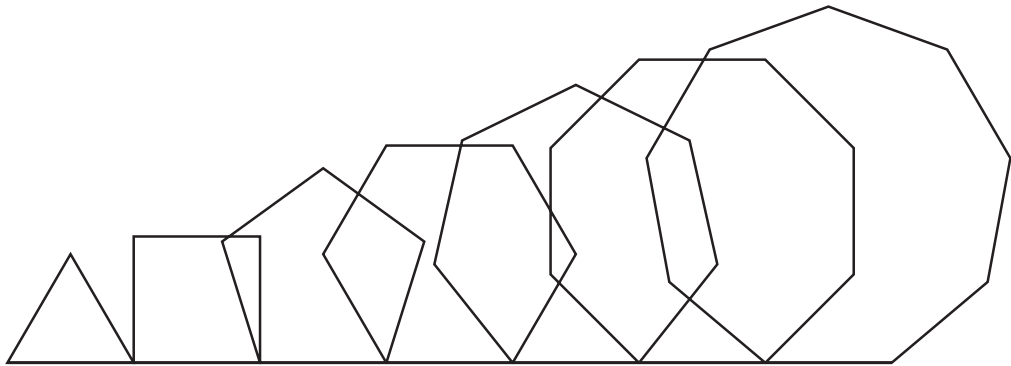
Figure 13-2. *Stars*

## Summary

- When you write a new method, it can call other methods.
- You can use a method without knowing how it is written; you need to know only what it does.
- After you finish writing a method, you can call it when you write other methods.
- Hiding the details of a method by giving it a name is called *abstraction*.



# Parameters and Arguments



In many previous scripts you sent messages with *arguments*. For example, in the message `go: 100` you specified that a robot should move a distance of 100 pixels, and the argument of this message is therefore 100. Although you have learned how to define methods, you have not yet learned how to define methods that require one or more arguments.

In this chapter you will learn how to define methods whose behavior depends on message argument values. We say that such methods have *parameters*, and that their behavior is parameterized. Method parameters act as placeholders in the definition of a method, and these placeholders are filled by message arguments when the message is sent. First, we will define a method with a parameter and invoke it. Then we will analyze it.

## What Is a Parameter?

The method `square` defined in Chapter 12 is rather limited, because the size of the square is fixed once and for all. You may have asked yourself, “what might be done so that I could draw a square with side length 300 pixels, or 175, or 225, or even 23 pixels?” There is nothing preventing you from defining the methods `square300`, `square175`, `square225`, `square23`, and so on.

But creating multiple square methods does not solve the problem in a satisfactory way. It would be very inconvenient if we had to define a new method every time we wanted to draw a square of a different size. What we would really like is to have a universal method for drawing squares that allowed the user to specify the side length at the time that we call the method. That way, we would not have to define a new method for every different square size.

What we need is to replace the fixed side length with a kind of variable whose value will be assigned when the message is sent, and not before. This kind of variable exists in many programming languages. It is called a *parameter*. A method parameter is a special variable that can take on an arbitrary value *at the moment the message is sent*. It therefore serves as a placeholder when you define the method and so is not given a value in the message definition.

This should all sound somewhat familiar. After all, you know that methods such as `go:` and `turnLeft:` take an argument (a distance or an angle) at the time the message is sent. Each time you wrote an expression such as `pica turnLeft: 90`, `pica turnLeft: 32`, or `daly turnLeft: 65`, you included in the message the value of the angle as an argument, and then the *method* `turnLeft:` used this value in deciding through what angle to turn the robot. In fact, in each of these expressions, it was the *same* method `turnLeft:` that was executed, each time with a *different value* for the angle through which `pica` or `daly` was to turn. Being able to specify different angles during different message sends using the same message selector `turnLeft:` is a powerful feature of the method `turnLeft:`. You would like to have an analogous feature for the method for drawing squares, as well as for other methods that you define. And you will, as soon as I explain how to define a method that like the method `turnLeft:` can take one or more arguments at execution time.

## A Method for Drawing Squares

You have seen that in Smalltalk, the name of a message selector is terminated by a colon (`:`) to indicate that the message takes an argument. Likewise, the name of the method corresponding to such a message selector also ends with a colon, and in the definition of the method, a parameter is used as a placeholder for the argument of a message. Thus if you want to create a method to draw a square of arbitrary size, you could use `square:` as its name and `sideLength` as the parameter. You can then define the method `square:` as shown in Method 14-1. This method is then used in Script 14-1, in which `pica` draws two squares, one of side length 10, and the other of side length 20.

**Method 14-1.** *The method square: uses the parameter sideLength to draw a square of arbitrary size.*

```
square: sideLength
 "Draw a square of the given side length"

 4 timesRepeat:
 [self go: sideLength.
 self turnLeft: 90]
```

**Script 14-1.** *The method square: is used to draw squares of different sizes.*

```
| pica |
pica := Bot new.
pica square: 10.
pica go: 300.
pica square: 20
```

Now let's analyze the definition of the method square: in Method 14-1. To define a method that requires one argument, the method name ends with a colon and is followed by the name of the parameter, which here is sideLength.

The parameter represents a variable whose value is defined when a message is sent (not when the method is defined). In Script 14-1, when the message square: 10 is sent, the parameter sideLength will be given the value 10. Then when the message square: 20 is sent, sideLength will have the value 20. Unlike regular variables in scripts, parameters are not explicitly declared using vertical bars | |.

In Method 14-1, the name of the parameter is sideLength. The name of the parameter should be chosen to indicate what it is used for. You could just as well have named this parameter length, as in Method 14-2 (or size, or anything else), as long as you replaced all the occurrences of sideLength in the method body with length. Note that Method 14-2 and Method 14-1 give *exactly* the same results. That is because sideLength and length are both names for exactly the same thing: the parameter for the method square:.

#### Method 14-2

```
square: length
 "Draw a square of the given side length"

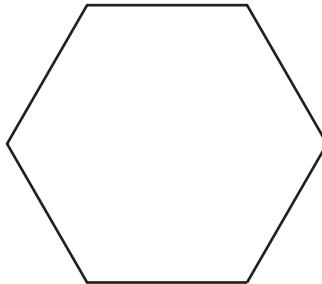
 4 timesRepeat:
 [self go: length.
 self turnLeft: 90]
```

## Practice with Parameters

Now it is time to practice a bit. Let's start with a simple exercise.

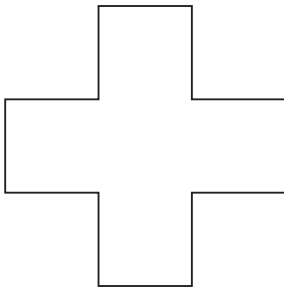
### Experiment 14-1 (A Method to Draw a Hexagon)

Define a method `hexagon`: that draws a hexagon with the side length passed as an argument.



### Experiment 14-2 (A Method to Draw a Cross)

Transform the script given below into a method named `cross`: that draws a cross with the length of one of its arms passed as argument. You should then be able to execute the expression `pica cross: 100`. Hint: notice that  $50 = 100 / 2$ . A good name for the parameter might be `armLength`.



```
|pica |
pica := Bot new.
4 timesRepeat:
 [pica go: 50.
 pica turnLeft: 90.
 pica go: 100.
 pica turnRight: 90.
 pica go: 100.
 pica turnRight: 90.
 pica go: 50]
```

## Variables in Methods

Just as we have used variables in our scripts to give names to certain quantities, we can also use variables in methods for the same purpose. If we wanted to tell `pica` to draw a polygon with side length 100, we might come up with something like Script 14-2. Because the value of the angle that `pica` has to turn through depends on the number of sides of the polygon, I have introduced the variables `numberOfSides` and `angle`. The number of sides is assigned a certain value (`numberOfSides := 6` in our example), and then `angle` is assigned a value that is calculated in terms of the value of `numberOfSides`. Now if we want to change the number of sides of the polygon from 6, as defined in the script, to any other value, we need to change only the value assigned to `numberOfSides` in the third line of the script, and we don't have to worry about the angle.

**Script 14-2.** *Drawing a polygon in a script using variables*

```
| pica numberOfSides angle |
pica := Bot new.
numberOfSides := 6.
angle := 360 / numberOfSides.
numberOfSides timesRepeat:
 [pica go: 100.
 pica turnLeft: angle]
```

To convert Script 14-2 into a method, we can define a parameter for the number of sides. This is done in Method 14-3, which defines the method `polygon100`: for drawing a polygon with an arbitrary number of sides, each side having a length of 100 pixels.

**Method 14-3.** *Drawing a polygon in a method using a variable and a parameter*

```
polygon100: numberOfSides
 "Draws a polygon with an arbitrary number of sides;
 the length of each side is 100 pixels"

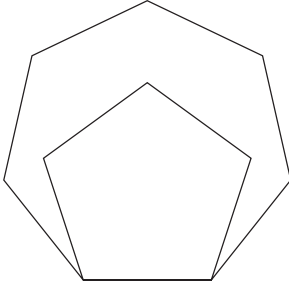
 | angle |
 angle := 360 / numberOfSides.
 numberOfSides timesRepeat:
 [self go: 100.
 self turnLeft: angle]
```

This method has one argument, `numberOfSides`, and one variable, `angle`. Both of them are used within the code of the method. Since `numberOfSides` is a *parameter*, its value is specified in the argument of any message that invokes the method, for example, `pica polygon100: 7` for a seven-sided regular polygon (heptagon). The *variable* `angle` is initialized in the text of the method by setting it to the necessary angle, which depends on the value of the parameter *at the time a message is sent* (so it will be `360 / 7` in our example). For any value of `numberOfSides`, the variable `angle` will have the correct value for a regular polygon with that number of sides.



Now that you have the method `polygon100:`, you can use it to draw polygons, as shown in Script 14-3.

**Script 14-3.** *Using the method `polygon100:` to draw a heptagon and a pentagon*



```
| pica berthe|
berthe := Bot new.
pica := Bot new.
berthe polygon100: 5.
pica polygon100: 7.
```

## Experimenting with Multiple Arguments

Why should we be limited to polygons with side length 100? Wouldn't it be better to have a method that draws an arbitrary regular polygon, where both the number of sides *and* the side length are determined when the message is sent? For that, we would need two parameters: `numberOfSides` and `sideLength`. So, how do we create a method having two parameters? You can create a method with two parameters by writing a method name with two colons and placing one argument name after each colon.

---

**Note** To define a method with multiple parameters, terminate each word in the method name (one word for each parameter) with a colon, and place each parameter after its corresponding word in the method name. The method named `polygon:size:` requires two arguments. The definition of the method `polygon: numberOfSides size: sizeValue` defines two parameters, `numberOfSides` and `sizeValue`. The first parameter represents, as its name implies, the number of sides. The second parameter is related to the size of the polygon. It will be explained following the definition of the method.

---

The definition of the method `polygon:size:` is shown as Method 14-3. After you have defined it, you can then simply send a message such as `pica polygon: 7 size: 100`.

#### Method 14-4

```

polygon: numberOfSides size: sizeValue
 "Draws a polygon with the number of sides and size to be specified"

 | angle sideLength |
 angle := 360 / numberOfSides.
 sideLength := 4 * sizeValue / numberOfSides.
 numberOfSides timesRepeat:
 [self go: sideLength.
 self turnLeft: angle]

```

You may wonder why the parameter `sizeValue` does not specify the side length of the polygon, but instead, I decided to define the side length (given by the variable `sideLength` in the method) to be  $4 * \text{sizeValue} / \text{numberOfSides}$ . In order to keep all polygons with the same `sizeValue` approximately the same size, I made all polygons have their perimeters equal to the perimeter of a square with side length `sizeValue`. The perimeter of such a square is  $4 * \text{sizeValue}$ . The result is that by setting the variable `sideLength` equal to  $4 * \text{sizeValue} / \text{numberOfSides}$ , when the robot draws `numberOfSides` sides each of length `sideLength`, it ends up walking a distance equal to the perimeter of a square with side length `sizeValue`. Thus for example, any polygon drawn with its second argument 100 will have perimeter 400 pixels, and so all the polygons will be displayed using about the same fraction of the screen.

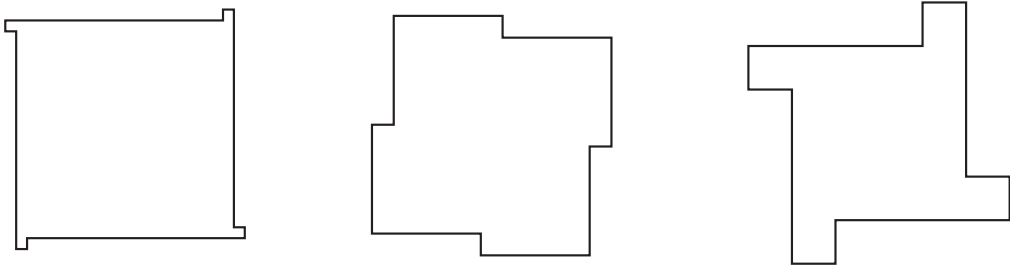
You may think that the name of the parameter `numberOfSides` is a bit too long and cumbersome. However, it is a very good name for the parameter, because it can easily be understood by any person reading the method. As we already discussed in Chapter 9, it is quite important that your code should be readable, almost like a story, by anyone. And that includes you: the name of a variable or parameter that is unclear may well stump you when you look at your code several months after you wrote it.

### Experiment 14-3

Define a method `rectangleWidth:height:` that draws a rectangle with its width and height passed as arguments.

### Experiment 14-4

By slightly modifying the method `cross`: that you wrote in Experiment 14-2, define a method `crossWalk1:walk2:` that can draw the stylized crosses shown in Figure 14-1. Order the parameters so that a normal cross like the one drawn by `cross`: will have its first parameter equal to twice the second, and so will be drawn by expressions such as `pica crossWalk1: 60 walk2: 30`.



**Figure 14-1.** Three stylized crosses are produced by the method `crossWalk1:walk2:`. The cross on the left is the result of the message send `pica crossWalk1: 5 walk2: 50`; the middle cross is from `pica crossWalk1: 50 walk2: 5`; the right-hand cross is from `pica crossWalk1: 10 walk2: 20`.

## Parameters and Variables

Now that you have practiced a bit, it is time to look more carefully at the difference between ordinary variables and parameters. Let's compare the script and method that were defined earlier for drawing a square of arbitrary side length. They are reproduced as Script 14-4 and Method 14-5.

In Script 14-4, first the variable `sideLength` is declared (line 1), then a value is assigned to it (line 3), and finally it is used as the argument of the method `go`: (line 5).

**Script 14-4.** *The square script using a variable*

```
(1) | pica sideLength |
(2) pica := Bot new.
(3) sideLength := 10.
(4) 4 timesRepeat:
(5) [pica go: sideLength.
(6) pica turnLeft: 90]
```

Method 14-5 shows examples of two features of parameters. First, the parameter `sideLength` is declared by being placed after a colon in the method name (line 1). Second, it is used as the argument of the message `go:` (line 5). A parameter is not initialized in the method definition because it always gets its value from the corresponding argument in any message that invokes the method. For example, when the message `pica square: 20` is sent to `pica`, then the parameter `sideLength` of Method 14-5 gets 20 as its value.

**Method 14-5.** *The square-drawing method using a parameter*

```
(1) square: sideLength
(2) "Draw a square of given side length"
(3)
(4) 4 timesRepeat:
(5) [self go: sideLength.
(6) self turnLeft: 90]
```

There are thus three distinctive differences between parameters and ordinary variables:

**Parameters are not explicitly declared.** Unlike other variables, a parameter does not have a variable declaration between vertical bars `| |`. A parameter is declared when it appears after a colon in the first line of the method's definition.

**Parameters cannot be assigned a value.** Parameters cannot be modified the way other variables can. You cannot assign new values to parameters inside the body of a method definition. For example, in Method 14-5 the expression `sideLength := 100` is impossible. Parameters cannot have their values modified because they are a special kind of variable. They are placeholders for the arguments that are passed when a message invokes the corresponding method, and their values are assigned by Squeak when a message is sent and thus cannot be explicitly assigned using `:=`.

**Variable initialization.** Ordinary variables and parameters get their values in very different ways. A variable value is changed by using an explicit assignment using `:=`. A parameter value is assigned when the method is invoked by a message. For example, the message `send pica square: 10` causes the parameter `sideLength` to be given the value 10. Thus a parameter is a variable, but it is a special type of variable whose value is assigned only at the time a message is sent and the corresponding method executed.

---

**Note** Aside from the three differences between parameters and ordinary variables, a parameter can be used in the code of a method definition just like any other variable.

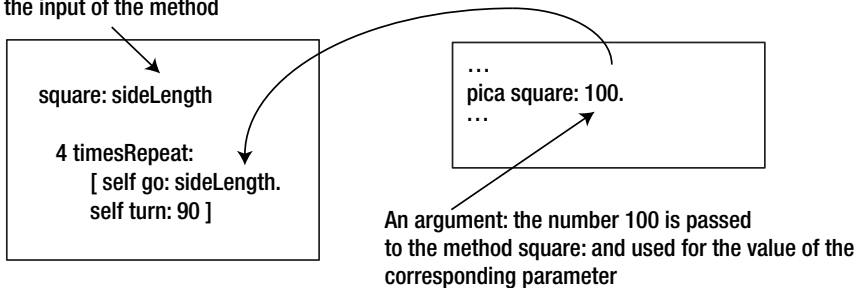
---

## Arguments and Parameters

I have introduced the two terms *argument* and *parameter* for two related but different ideas. An argument is a specific object passed in a message. A parameter is the placeholder variable used in a method definition whose precise value isn't known when the method is defined. A parameter takes its value from a corresponding argument.<sup>1</sup>

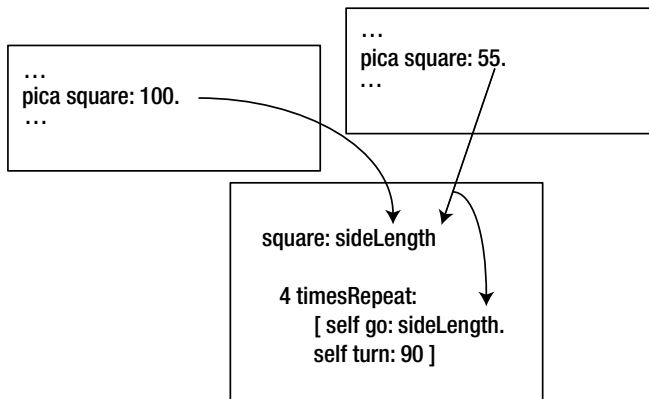
In Figure 14-2, in the message `square: 100`, the number 100 is the message argument. When the method `square:` is executed, its parameter `sideLength` is set to 100, the value of the argument.

A parameter: the placeholder variable `sideLength` represents the input of the method



**Figure 14-2.** The relationship between an argument (an object) and a parameter (a placeholder variable)

Another way to understand the difference between an argument and a parameter is that a parameter is a placeholder inside a method that represents an input to the method, while an argument is the actual value that is passed as this input. This idea is illustrated in Figure 14-3.



**Figure 14-3.** The value of the argument is bound to the parameter during execution of the method.

1. Many authors define these terms differently. Some use “actual parameter” for what we call “argument” and “formal parameter” for what we call “parameter.” Others use the terms “parameter” and “argument” interchangeably.

Note that a parameter can also be used as an argument in other message sends. For example, in the definition of the method `square:` (Method 14-5), the parameter `sideLength` is used as the argument in the message `go: sideLength`.

A message argument can also be a variable. For example, in Script 14-5, which uses the method `square:`, the argument of the first message `square:` is the value of the variable `squareSize`, which is 100. The argument of the second message `square:` is the value of the expression `squareSize + 200`, which is 300. The parameter `sideLength` of the method `square:` gets the value 100 from the first `square:` message, and then the value 300 from the second `square:` message.

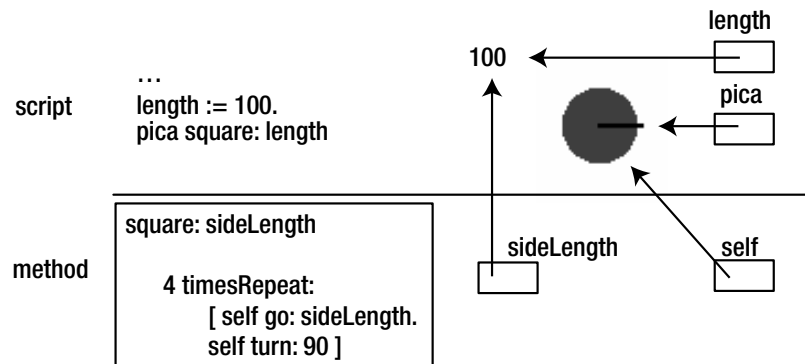
**Script 14-5.** *A variable as argument*

```
| pica dist |
pica := Bot new.
squareSize := 100.
pica square: squareSize.
pica go: 300.
pica square: squareSize + 200
```

## About Method Execution

On a first reading you may wish to skip this section, since it goes into details that beginners do not need to know. I wrote it because I wanted to answer the questions of the most curious readers, but I could as easily have omitted this paragraph without loss of continuity.

When a method is executed, certain new variables are created. These variables are the message receiver `self` and the method parameters (which refer to the method arguments), such as `sideLength` in Figure 14-4, which shows the effect of sending the message `square: length` to a robot referred to by the variable `pica`, where the variable `length` references the number 100.



**Figure 14-4.** *When a message is sent and a method executed, new variables are created that refer to the arguments and the receiver of the message.*

When the method `square:` is executed, the variable `self` refers to the message receiver, which in our example is the robot pointed to by the variable `pica`; and the parameter `sideLength` refers to the value of the variable `length`, which here is the number 100. The same process occurs for each message send. For example, the execution of the expression `daly square: 200` assigns to `self` the robot referenced by the variable `daly` and assigns to `sideLength` the number 200.

This may look complex, but you do not have to worry about it. These are the hidden steps that Squeak takes to make sure that parameters are set to the values of the message arguments.

## Summary

- A parameter is a special kind of variable that acts as a placeholder for message arguments. A method parameter is declared right after a colon in the method name indicating the position of the parameter. A parameter must not be declared as a variable, and it cannot be assigned a value in the body of a method definition. Parameters receive their values from message arguments when a message invokes the method.
- To define a method with multiple arguments, terminate each word in the method name with a colon, and place each parameter after its corresponding word in the method name. For example, the method named `polygon:size:` requires two arguments. The definition of the method `polygon: numberOfSides size: sizeValue` defines two parameters, `numberOfSides` and `sizeValue`.



# Errors and Debugging

**N**ow that you know how to define methods that call other methods, you will be able to write more and more complex programs, and sooner or later you will find that you have made an error and cannot figure out what is wrong. Errors in computer programs are called *bugs*, and I am going to show you a powerful tool that helps you to find those bugs and get rid of them: the Squeak debugger.

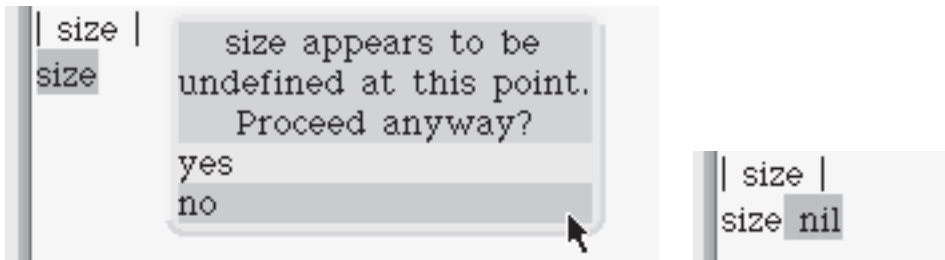
A *debugger* is a tool that shows the execution of a program. It lets you inspect and change the values of the variables and edit the methods of a program. In this chapter I will present some typical examples of errors and explain what a debugger is and how to use it. I will begin by presenting some common errors regarding variables. Then I will show you how to use the debugger to identify and fix other kinds of problems.



## The Default Value of a Variable

Variables are very useful in programming, but they require a bit of attention. For example, you can easily introduce a bug in a program by using a variable that has not been declared or that has been assigned incorrect values. Even experienced programmers make errors. However, an experienced programmer knows how to find and fix bugs. Squeak provides some help by checking, for example, whether the variables that you use have been declared. Such structural, or *syntax*, errors in a program are easy to catch, and Squeak will catch them. However, Squeak has no way of detecting *logical* errors, such as assigning a variable the value 1000 when you should have assigned it 100. Such errors you will have to find yourself, and using a debugger can help you to understand, locate, and fix your errors.

First, let's experiment a bit. Type, select, and print Script 15-1, which declares the variable `size` and then attempts to use it before it has been initialized. You should obtain Figure 15-1, which shows that Squeak prompts you when a variable is not initialized. If you selected the choice **yes** when prompted when you print Script 15-1, you should get `nil` printed as shown by Figure 15-1 (right). The value `nil` that you just obtained is a special object assigned to any declared variable. We will begin by looking at the default value of a variable.



**Figure 15-1.** Left: Squeak prompts you when you attempt to use a variable that has not been initialized. Right: The value `nil` is printed when you proceed.

**Script 15-1.** Attempting to use a variable before it is initialized

```
| size |
size
```

The reason that Squeak squawks when you use an uninitialized variable is that if you do not initialize one or more variables, your program is almost certain to run incorrectly, because your program will be using a variable that has an incorrect value or even no value at all. In Smalltalk, the value of a variable is by default the object `nil`, which represents an undefined value. Although `nil` is an object like any other, it does not understand many messages. In particular, it does not understand any of the messages that are sent to number objects or to robot objects. Therefore, you will get an error if you try to send such a message to the object `nil`. The object `nil` is important for determining whether a variable has been assigned a value.

---

**Important!** In Smalltalk, the value of an uninitialized variable is by default the object `nil`, which is used to represent an undefined value.

---

The error occurring during the execution of Script 15-1 generates the message shown in Figure 15-1. Normally, just answer **no** and go back and initialize the variable. However, although Squeak is able to analyze your scripts for such structural errors, it cannot detect before a script is run whether you have used an *invalid* value. In such a case, Squeak opens a *debugger*, which you can use to access the execution state, which includes the receiver of the message, the message itself, the variables that are available, and so on. This is what I will explain now.

## Looking at Message Execution

In case you have forgotten the definitions of the methods `pattern` and `pattern4`, they are re-created as Methods 15-1 and 15-2. What is important to note here is that the method `pattern4` uses the method `pattern`, while the method `pattern` in turn invokes the methods `go:` and `turnRight:`.

### Method 15-1

#### `pattern`

```
"draws a pattern"
```

```
self go: 100.
self turnRight: 90.
self go: 100.
self turnRight: 90.
self go: 50.
self turnRight: 90.
self go: 50.
self turnRight: 90.
self go: 100.
self turnRight: 90.
self go: 25.
self turnRight: 90.
self go: 25.
self turnRight: 90.
self go: 50
```

**Method 15-2****pattern4**

```
"draws four patterns"
```

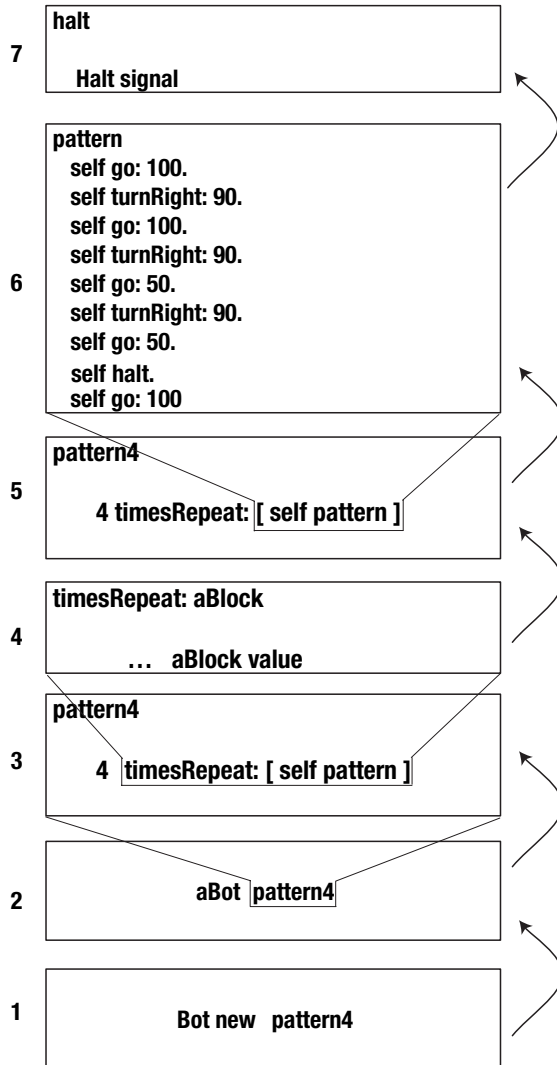
```
4 timesRepeat: [self pattern]
```

After these methods have been defined, if Squeak executes the expression `Bot new pattern4`, several messages get invoked in a chain reaction whose end result is that the robot draws four patterns on the screen. Let's have a look at this chain of messages: First, `Bot new` creates a new robot, to which the message `pattern4` is sent. As a result, the method `pattern4` is executed. This execution sends the message `timesRepeat:`, which in turn sends the message `pattern`. The execution of the method `pattern` sends several messages, namely `go:` and `turnRight:`. In programming language jargon such a chain of messages is called an execution stack: the stack contains all the methods executed in reaction to an initial message and the chain of messages that those methods executed, then all the methods executed in reaction to each of this second set of messages and the chain of messages that those methods executed, and so on until there are no more messages left.

A possible representation is shown in Figure 15-2, where the last method to be executed is on top, and so on down the line. A method that calls another method appears below the called method. The part of each expression that leads to the invocation of the method above it appears in boldface type. Let us look at what is going on in detail.

1. In the bottom box, the message `new` is sent to the class `Bot`, which creates a new robot.
2. The message `pattern4` is sent to the created robot.
3. The execution of the method `pattern4` sends the message `timesRepeat: [self pattern]`. The message `timesRepeat:` is in bold, since it is the first to be sent.
4. The execution of the method `timesRepeat:` leads to the execution of the block. This is done by sending the message `value` to the argument of the message `timesRepeat:`.
5. In the method `pattern4` the message `pattern` is sent by the method `timesRepeat:` as a result of the block execution.
6. The process continues in a similar manner, with the messages of the method `pattern` being executed one after another.

Figure 15-2 contains one other box, which I will explain in a minute. What you should understand at this point is that one method calls another one and that the called method is above the calling method on the stack.



**Figure 15-2.** The execution stack, which contains all the messages and the methods invoked as a result of the execution of Bot new pattern4

## A First Look at the Debugger

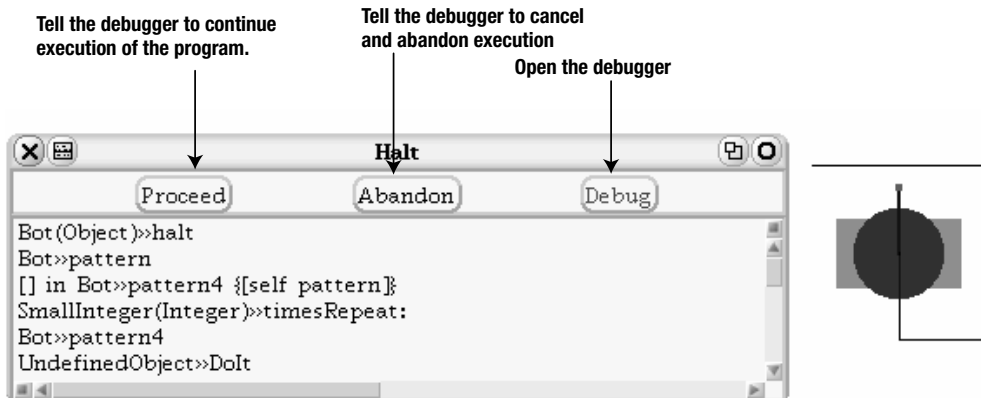
Figure 15-2 shows how you can picture the sequence of messages sent and methods executed resulting from the execution of a message. The Squeak debugger actually lets you see, navigate, and change the chain of messages. The debugger is automatically invoked when a message is not understood by an object, as I will show later, but you can invoke it explicitly by introducing the expression `self halt` in the body of a method. Introducing such an expression is useful when you want to understand how an expression is executed or to locate a bug in a program.

---

**Important!** The Squeak debugger is a tool that allows you to navigate through a sequence of executed methods. Using a debugger, you can print the values of a method's arguments, modify the method's definition, change and view the values of variables and arguments, and continue execution.

---

To open the debugger, add the expression `self halt` in the method pattern, as shown in Method 15-3. Then execute the expression `Bot new pattern4`. You should obtain the situation depicted in Figure 15-3. First a new robot is created. Then it starts to draw the first pattern, and after drawing four lines, it stops, and a window opens that gives you an opportunity to open the debugger.



**Figure 15-3.** The robot starts to draw the beginning of the first pattern, but after drawing four lines, it stops, and you can open the debugger.

**Method 15-3.** *Introducing the expression `self halt` in a method opens a window that gives you the opportunity to open the debugger.*

**pattern**

```
"draws a pattern"

self go: 100.
self turnRight: 90.
self go: 100.
self turnRight: 90.
self go: 50.
self turnRight: 90.
self go: 50.
self halt.
self turnRight: 90.
self go: 100.
self turnRight: 90.
self go: 25.
self turnRight: 90.
self go: 25.
self turnRight: 90.
self go: 50
```

---

**Important!** To invoke the debugger, insert the expression `self halt` in a method. The execution of the expression `self halt` opens a window that gives you an opportunity to open the debugger.

---

The debugger window shown in Figure 15-3 offers three buttons: **Proceed**, **Abandon**, and **Debug**.

**Proceed.** This button tells the debugger to continue execution of the method, ignoring the `self halt` message. Note that it is possible to proceed only if you opened the debugger using `self halt`. When an actual error occurs that opens the debugger, using **Proceed** is of no help, since Squeak cannot continue.

**Abandon.** This button tells the debugger to close, and execution is simply discontinued.

**Debug.** This button tells the debugger to open. An open debugger window is shown in Figure 15-4.

If you press **debug** and select the second line of the top pane, you obtain the window shown in Figure 15-4. As you can see, the debugger is composed of several panes. The top pane represents the execution stack, as was outlined in Figure 15-2. You are shown all the messages that have been sent up to just before the halt occurred, with the most recent message on top. The most recent message is of course `halt`, and therefore it is at the top of the stack. This is the message that led to the opening of the dialog box shown in Figure 15-3.

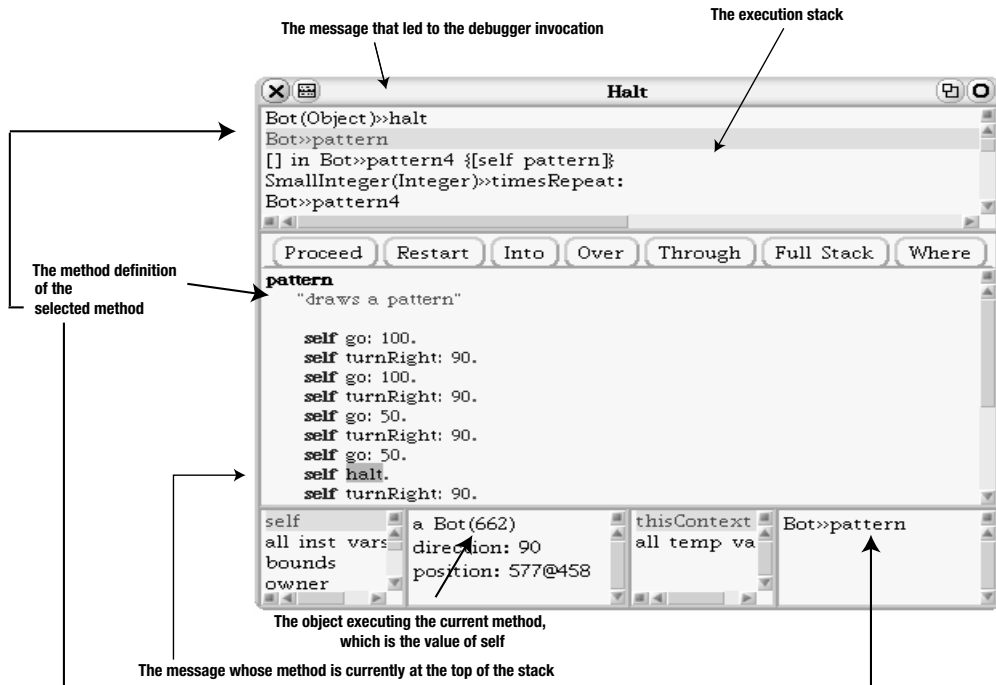


Figure 15-4. The debugger window. The method `pattern` is selected.

Selecting one of the lines in the top pane shows the method definition in the next pane down. Figure 15-4 shows that we selected the method `pattern` in the stack, and its definition is shown in the second large pane. The object that received the message `pattern` (which is the object referred to by the variable `self`) and executed it is shown in the bottom left pane.

In the method body of the currently selected method, which in our example is `pattern`, the debugger highlights in green the method whose execution is currently stopped, and which is above the selected method on the stack. Here `self halt.` is currently stopped, and it is above `pattern` on the stack. You can also see that all the expressions above `self halt.` in the method body have been executed, while the expressions below have not. In this example, `self go: 100.`, `self turnRight: 90.`, `self go: 100.`, `self turnRight: 90.` have already been executed.

If you select the third line in the top pane, that is, the method `pattern4`, you will see, as shown in the left pane of Figure 15-5, that the stack of the execution of the method `pattern4` above is related to the execution of the expression `self pattern` of the method `pattern4`. Now if you select the fifth line, you have another look at the method `pattern4`, but now before the `timesRepeat: loop` is executed.

If you select the fourth line, as shown in Figure 15-5, you see the definition of the method `timesRepeat:` itself. You can see in the top pane that the object receiving the `timesRepeat:` message is not a robot but an integer. In our example, in the expression `4 timesRepeat: [ self pattern ]` the receiver of the loop is the integer object 4. This means that within this method the variable `self` is bound to an integer object. This is nothing unusual, since the variable `self` *always* represents the object that has received the current message.

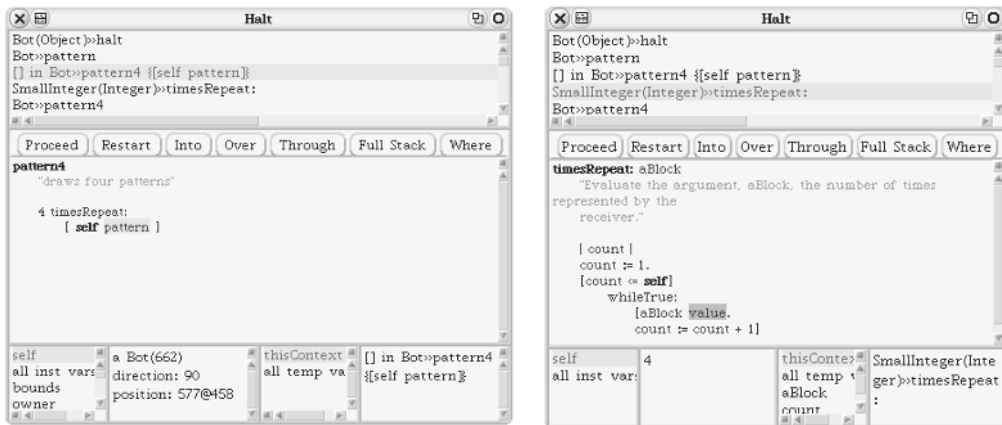


Figure 15-5. Left: The method `pattern4` is selected. Right: The method `timesRepeat:` is selected.

## Stepping through the Stack

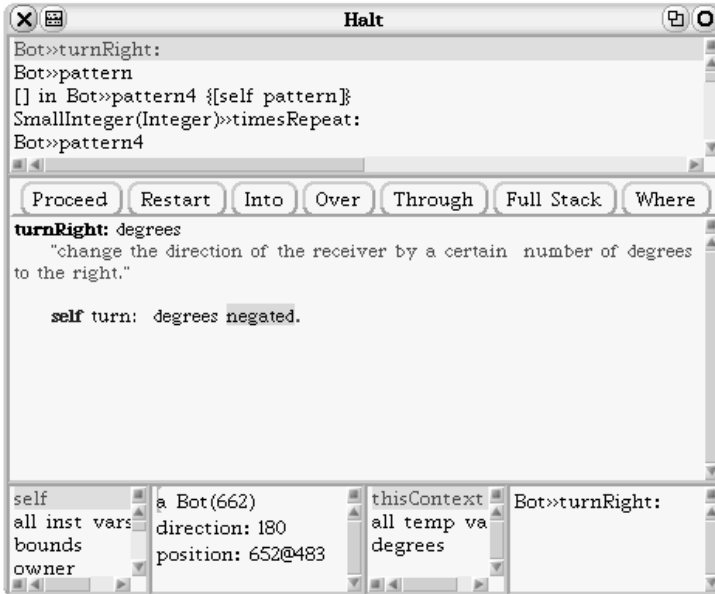
The Squeak debugger not only lets you navigate the stack and identify the receivers of the various messages, but lets you execute the method step by step. You can tell the debugger to perform several actions using the buttons located between the first and second panes. Here is a description of the most useful ones, in order from left to right.

**Proceed.** Pressing this button has the same effect as pressing the **Proceed** button in the dialog box shown in Figure 15-3. The debugger is closed, and execution of the method continues if possible.

**Restart.** You can ask the debugger to restart the execution of the current method. Note that sometimes, doing this can lead to difficulties, since you might thereby modify the same object twice, leading to an unexpected result or an outright error.



**Into.** This button and the next one are the most useful buttons in the debugger and the most frequently used. Pressing this button takes you into the method currently selected without executing it. That is, you are taken inside the code for the selected method as displayed in the second pane of the debugger, and no execution of the method takes place. See Figure 15-6 for an example, in which the method `turnRight:` of the method `pattern` is selected and stepped into.

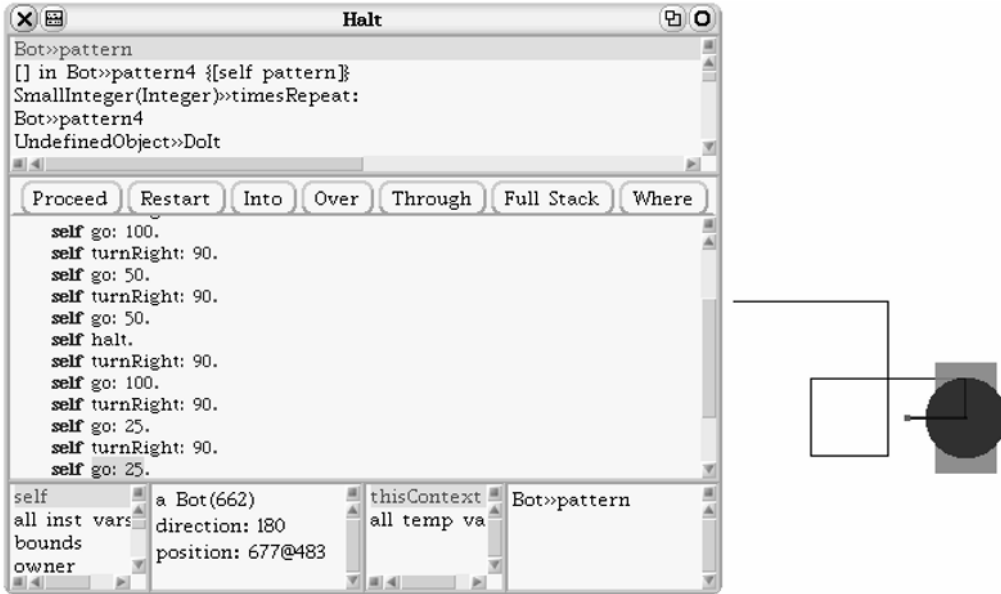


**Figure 15-6.** Stepping into the method `turnRight:`.

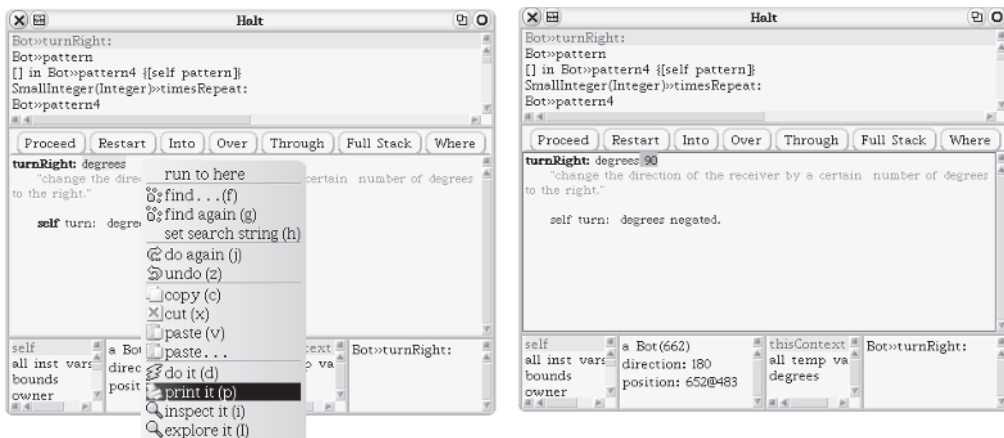
**Over.** Pushing this button lets you execute the message currently selected without stepping into the method. You simply execute the expression and then stop. See Figure 15-7. In this figure, I stepped *into* the method `pattern`, and then I stepped *over* some expressions in the method `pattern`, which led to those methods being executed. You can see in the figure that I have stopped at the expression `go: 25`. I am still inside the method `pattern` because I did not step into any of the expressions that constitute `pattern`. As the methods are executed, you should see the robot performing each action one after the other. Note that when you arrive at the end of the method, having executed all of the expressions, stepping over just returns you to the method that invoked the current one, and you move down the stack. For example, after you were done stepping through `pattern`, you would end up in `pattern4`.

When you press the button **Into**, you are asking the debugger to go *into* the method without executing it. For example, if the currently selected expression in the method `pattern` is `self turnRight: 90`, then you can either simply execute that expression by pressing the **Over** button, or you can press the **Into** button, which tells the debugger to go inside the method `turnRight:`, stopping at its first expression, as shown in Figure 15-6, where as you can see, the first message to be sent is the message `negated`. And here again you have the choice to step

into the expression negated or to execute it by stepping over it. And once again, when you reach the end of a method, you return to its calling method. Note that you can also see the value of the arguments passed to the method by selecting the argument name on the method body and choosing **print it** (see Figure 15-8).



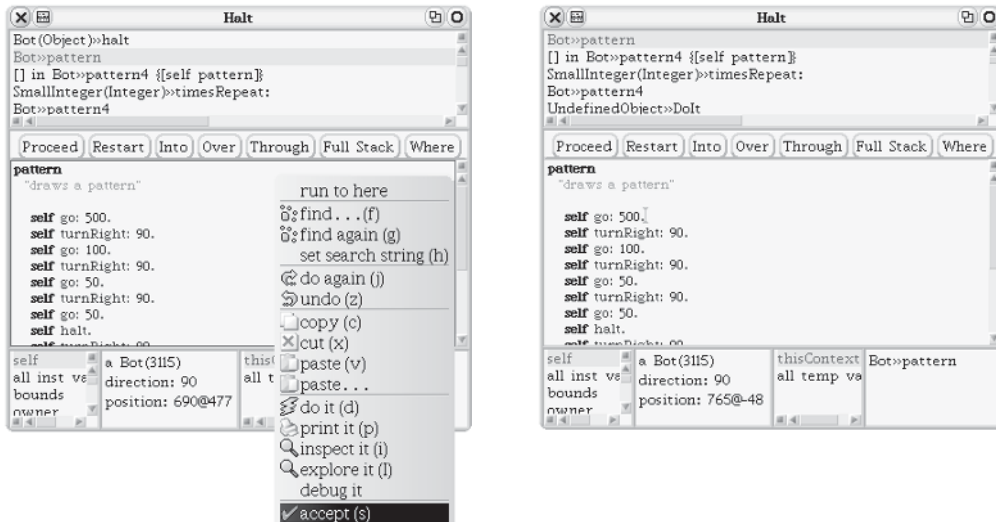
**Figure 15-7.** Stepping in the method pattern and then stepping over each message, watching the robot executing each expression one by one.



**Figure 15-8.** Left: Selecting an argument and printing. Right: The value is printed.

Finally, you can *modify* the method definition inside the debugger by editing the code and then accepting it via the contextual menu item **accept** (see Figure 15-9). For example, in Figure 15-9, I restarted the method by pressing the **Restart** button; I then edited the method in the debugger itself, replacing the 100 of the first `go:` message with 500. Then I recompiled the method by choosing **accept** in the menu. Now the robot will move 500 pixels if you press the **proceed** button.

This is all somewhat complex, and you might find it confusing at first. Remember that the debugger simply lets you execute all the expressions of a method step by step. So please experiment with the debugger and try out all the buttons. Be assured that no robots will be harmed in the process.



**Figure 15-9.** Left: Editing the method definition and recompiling it. Right: The method `pattern` has been recompiled.

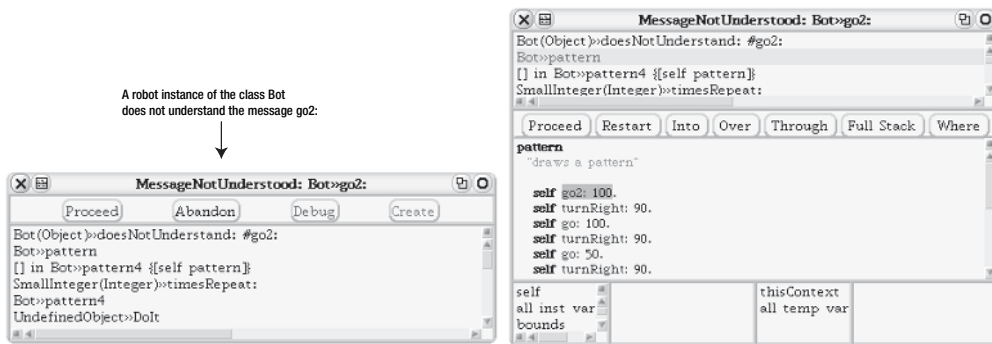
## Fixing Errors

I have shown you how to open the debugger by inserting the expression `self halt` in a method. I have also shown how you can edit the code of a method in the debugger. This technique allows you to use the debugger to fix your errors, that is, to *debug* your code. When an object receives a message that it does not understand, you have the opportunity to open the debugger. When an object does not understand a message, Squeak sends this object the message `doesNotUnderstand:` with a representation of the message. By default, the method `doesNotUnderstand:` opens a dialog box to ask you whether you want to open the debugger. You can then use the debugger to navigate in the stack of executed methods and try to understand what went wrong.

## Example 1

To illustrate the process, change the first line of the method `pattern` to `self go2: 100` and execute the expression `Bot new pattern4`. You should obtain the debugger dialog box, as shown in Figure 15-10. The debugger dialog box indicates that the receiver, a robot created by the class `Bot`, does not understand the message `go2:`. When you press the button **Debug** in the dialog box, you open the debugger, as shown in the right pane of Figure 15-10.

The method on the top of the stack is the method `doesNotUnderstand:`, which is sent to the receiver of a message when that receiver does not understand the message. The method directly below it is then the method containing the message leading to the error and hence the call to the method `doesNotUnderstand:`. Here the method `pattern` contains the message `go2:`, which is not understood by the robot, as shown in Figure 15-10.



**Figure 15-10.** *Left: A doesNotUnderstand: error has occurred. Right: Identifying the problem.*

## Example 2

Now change the first line of the method `pattern` to `self go: nil` and execute the expression `Bot new pattern4`. You should get the debugger dialog box shown in Figure 15-11. The error is difficult to spot. Here, the dialog box title `MessageNotUnderstood: UndefinedObject` indicates that there is a message not understood sent to `nil`, which is an instance of the class `UndefinedObject`.

The fact that `nil` was passed as a value led to an error after several other method executions. Therefore, you have to go down the stack to the point where you can understand your mistake and fix it. For example, in the right-hand pane of Figure 15-11, the second method from the top shows that something wrong happened with `*`, but the problem does not come from this method. This is the same situation in the following method, `go:`. If you select that method, you can see that the method `go:` just passes the argument it receives from the method `pattern` to the method `positionIfGo:`. If you select the parameter `distance` and print its value, you get `nil`. This indicates that the problem comes from a place further down the stack.

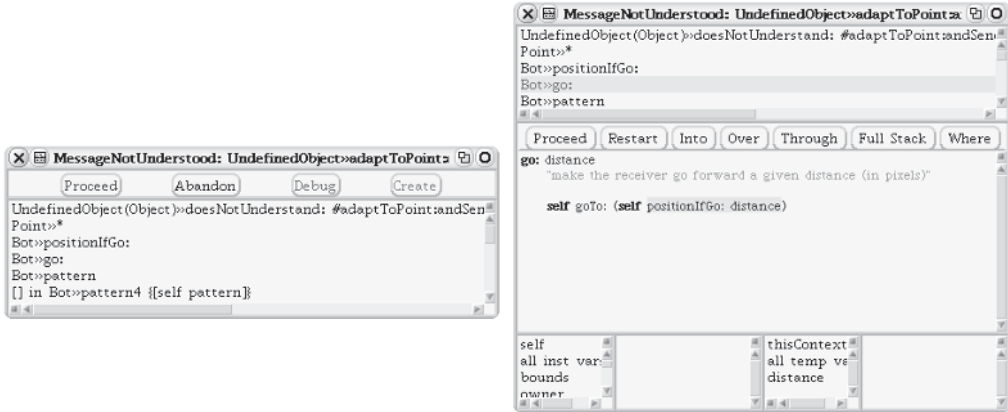


Figure 15-11. Left: A message not understood sent to nil. Right: Going down the execution stack.

Finally, from the left pane of Figure 15-12, you can see that nil is passed as an argument instead of a number, as expected by the method go:. You can now fix the bug by editing the method pattern to change self go: nil to self go: 100, accepting all the changes using the pop-up menu, and pressing the button **Proceed** to continue execution.

This example of using nil as the argument in a go: message provides a very easy problem to identify. While you may occasionally introduce such trivial bugs, you will generally have to face a wide variety of unexpected situations that lead to bugs. However, the process is the same: you use the debugger to navigate through the stack, checking the values of the arguments until you understand what went wrong. You may find that your code itself needs to be changed, not only the arguments of messages.

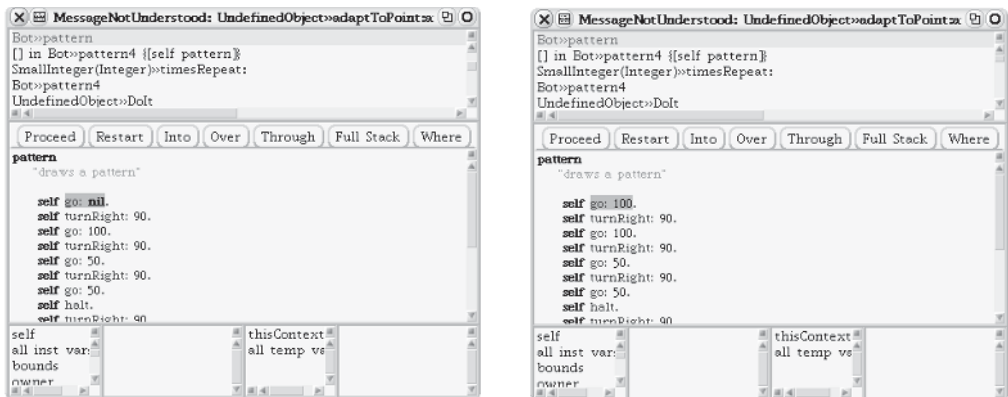


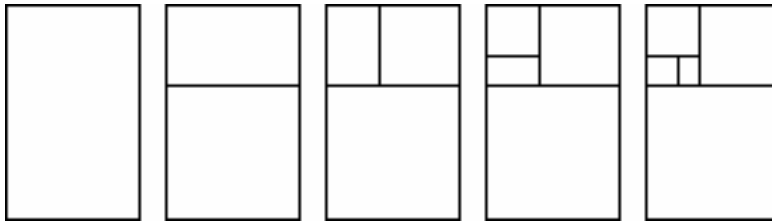
Figure 15-12. Left: Editing the method definition and recompiling it. Right: The method pattern has been recompiled.

## Summary

- In Smalltalk, the value of an uninitialized variable is by default the object `nil`, which is used to represent an undefined value.
- The Squeak debugger is a tool that allows you to navigate through the executed methods. Using the debugger, you can print the values of arguments, modify the definition of methods, and continue execution.
- To open the debugger explicitly, insert the expression `self halt` in a method. When the expression `self halt` is executed, the debugger can be opened.



# Decomposing to Recompose



In this chapter I would like to show you that a good approach to solving a problem is to *decompose* the problem into smaller problems that are easier to solve, and then *compose* the solutions of the smaller problems to solve the main one. Often, defining and using a number of simple methods lets you solve a complex problem in a much simpler and more natural way than by attempting to solve the whole thing at once. This is yet another example of the power of abstraction that I introduced in Chapter 12.

Unfortunately, although the decomposition of problems into smaller ones is a powerful technique, there is no systematic way to decompose a problem. Only through experience and trial and error is it possible to learn and develop some intuition about problem-solving. In this chapter, I offer you some small but nontrivial problems that you should try to solve. That is, try to decompose each problem into smaller problems and then compose your solutions to form a solution to the initial problem. The problems proposed here are certainly the most complex exercises in this book, so do not be discouraged if your first attempts fail. It is important, though, that you try to solve them.

## Mazes and Spirals

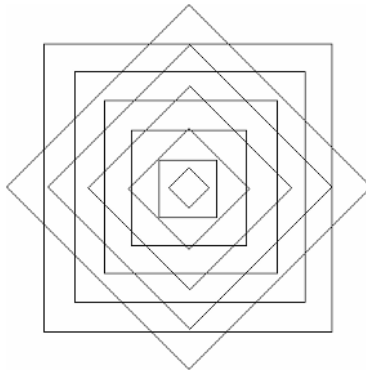
In Chapter 10, you experimented with loops and variables. At that time, you didn't know that you could use methods that you wrote yourself. Now I suggest that you redo those experiments using methods such as `square:` and that you define methods with multiple arguments to help you. As I already mentioned, solving complex problems using simpler methods is a difficult task that can be learned only by experience. Therefore, I strongly encourage you to do all the experiments in this chapter and indeed, you should try to find more than one solution for each of them. Try to see how the solution to a problem can be expressed at a high level of abstraction. Such an approach makes the overall problem simpler to express. For example, to draw a pyramid of squares, you could simply draw a large number of squares. But at a higher level, you could view the pyramid as a number of rows of squares of varying lengths, and then try to define and use a method that draws a row of  $n$  squares.

## Centered Squares

The following experiments give you more opportunities to practice your skills. Imagine different strategies to solve these problems. Here is a hint: define a method `centeredSquare: size` that draws a square centered around the robot, and after drawing the square brings the robot back to its original position.

### Experiment 16-1 (Square Ripples in a Square Pond)

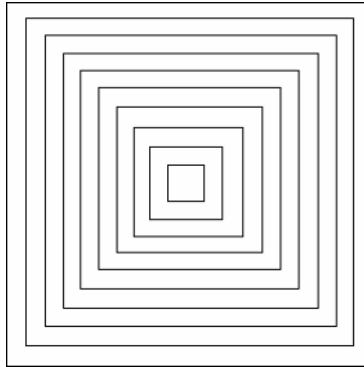
By composing methods, draw the picture shown below.



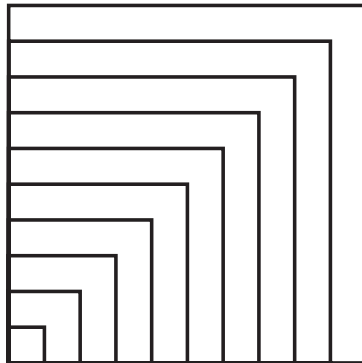


**Experiment 16-2 (A Corridor)**

By composing methods, draw the picture shown below.

**Experiment 16-3 (Russian Squares)**

Use the method square: to build squares of different sizes as shown in the figure below.



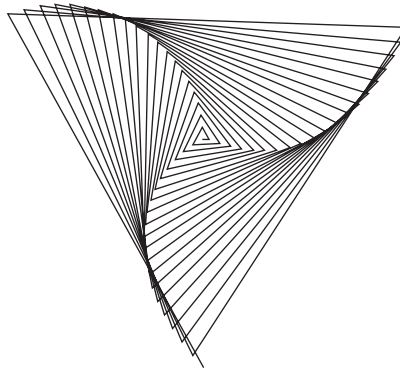
## Spirals

The spirals in this chapter are all drawn using just straight segments, angles, and repetition. They have no curves; the robot draws a straight line segment as it goes forward, then turns through an angle and repeats the process.

In the first group of spirals, the length of each segment changes a little from the previous one. The robot turns through the same angle after each straight segment. These are called *constant-angle spirals* because the angle doesn't change.

### Experiment 16-4 (A Constant-Angle Spiral)

Define a method `constantAngleSpiral`: that draws a constant-angle spiral. The angle through which the robot turns is the sole parameter. Change the distance through which the robot travels by the same amount for each segment. Have fun and try different angle values; you can create wonderful drawings. The following script created the spiral shown below.



```
| pica |
pica := Bot new.
pica constantAngleSpiral: 121
```

### Experiment 16-5 (Another Spiral)

In the previous experiment, at each step you added the same number to the distance through which the robot moves. Now experiment by computing the length of a side, not by adding a constant amount to the previous side length, but instead by multiplying by a constant ratio. Pay attention! Multiplying by 1.1 means an increase of 10 percent.

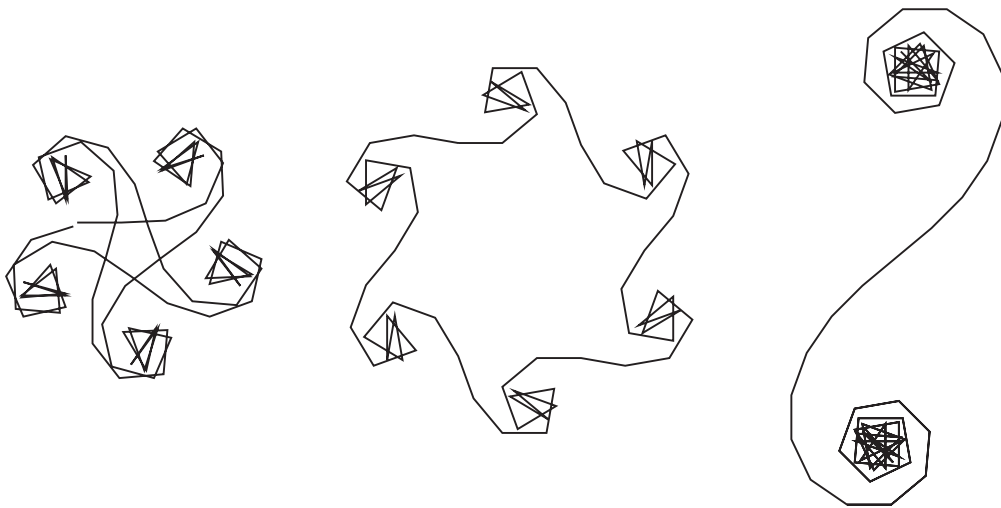
### Experiment 16-6 (A Spiral with Four Parameters)

The `constantAngleSpiral` method depends on four different values. They are the *number of segments*, the *starting length* for the first segment, the *amount by which each segment is incremented*, and the *angle* through which the robot turns. Define a method named `spiralSegments:firstLength:lengthIncrement:constantAngle:` that can draw all of these spirals by varying the choice of four parameters. Try, for example, the following script, which draws two different spirals.

```
| pica |
pica := Bot new.
pica spiralSegments: 50 firstLength: 10 lengthIncrement: 3 constantAngle: 144.
pica color: Color red.
pica spiralSegments: 120 firstLength: 1 lengthIncrement: 3 constantAngle: 12.
```

### Experiment 16-7 (Spirals with Constant Distance)

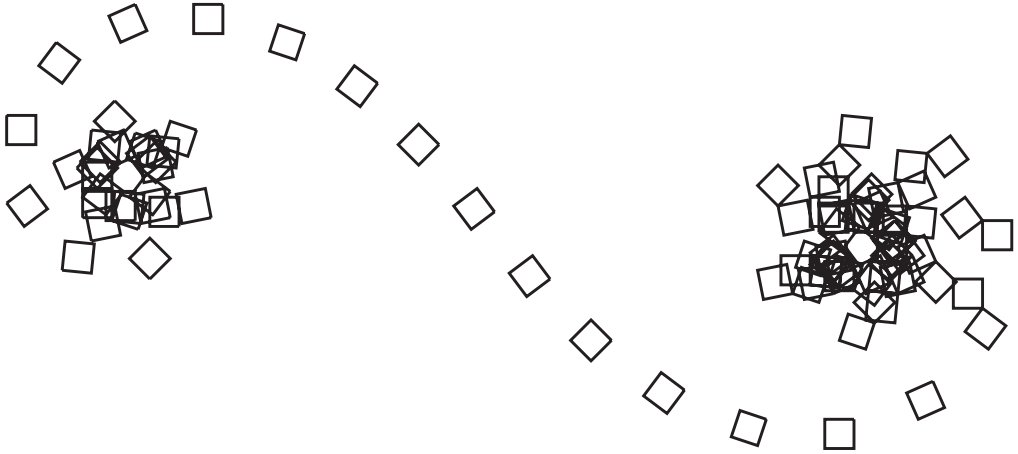
Up until now, you created spirals by *changing the distance* through which the robot moved forward, and the robot always turned *through the same angle*. Now experiment with doing the opposite: keep the distance constant, and increase the angle by a fixed increment after each segment. For these experiments, define a method with four arguments: the number of times to repeat, the initial value of the angle, the angle increment, and the length of a segment. As shown in Figure 16-1, which depicts three of these spirals, predicting the curves generated by this method is quite difficult. Feel free to play with different values. Have fun!



**Figure 16-1.** Left: Iterations: 90, initial angle: 2, increment: 20, length: 30. Middle: iterations: 72, initial angle: 40, increment: 30, length: 30. Right: iterations: 100, initial angle: 40, increment 5, length: 23.

### Experiment 16-8 (“Spirals” Out of Squares)

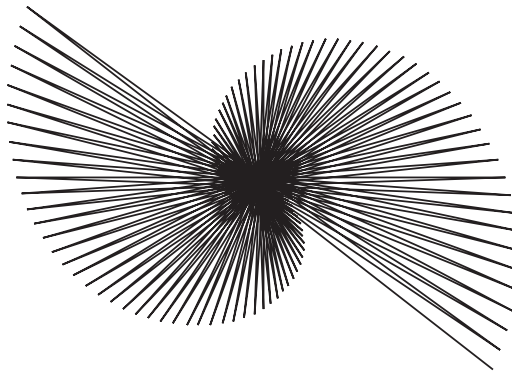
If you change the method that you defined in Experiment 16-7 to draw a small square instead of a line, you can get really crazy pictures like the one shown in Figure 16-2.



**Figure 16-2.** Drawing a fixed-length “spiral” using squares

### Experiment 16-9 (A Spiral Out of Lines)

Try to reproduce the figure shown below. Here are some hints: start with a length of 5 and increase it by 3, and turn through an angle of 178 degrees.



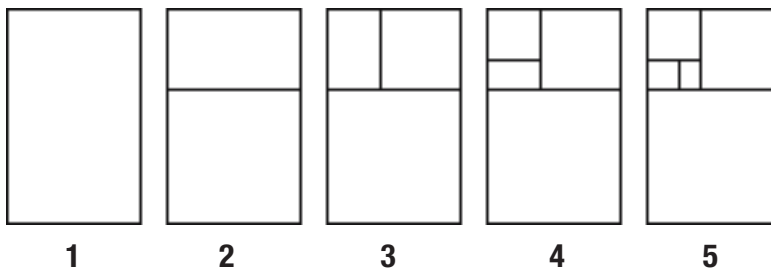
## Golden Rectangles

Now it is time to write some methods for drawing the famous golden rectangles that were introduced in Chapter 8. I encourage you to try your own methods before reading my solution. For some of the experiments I do not offer a solution. Instead, I just give you some hints.

A golden rectangle is a rectangle with the property that if you cut a square out of one end of the rectangle with side length equal to the rectangle's width, the rectangle that is left over is again a golden rectangle. For example, rectangle 2 in Figure 16-3 comprises a square on the bottom and another golden rectangle on the top. In rectangle 3, the golden rectangle inside rectangle 2 has been divided into a square and an even smaller golden rectangle. The process is repeated again in rectangles 4 and 5.

It is not difficult to show that for a rectangle to have this property, the ratio of the length to the width must be  $\frac{1+\sqrt{5}}{2}$  (which is approximately equal to 1.6).<sup>1</sup> This special number is called the *golden ratio* or *golden section*. It can be calculated in Smalltalk by the expression `1 + 5 sqrt / 2`.

The problem that I am posing for you is to draw several golden rectangles, each one nested inside the previous one, as shown in Figure 16-3.



**Figure 16-3.** The first five steps in the construction of nested golden rectangles

Try to identify actions that might help you achieve your goal. To help yourself, take a piece of paper and draw a golden rectangle of width about 10 centimeters with one or two golden rectangles inside, like those in Figure 16-3. For example, you might first try to define a method `goldenRectangle`: that draws a golden rectangle. Then you could define a method `manyGoldenRectangles`: that draws several golden rectangles by calling the previous method.

Another approach is to draw one golden rectangle and then draw just one line inside that rectangle to create the next golden rectangle, and so on. Don't get discouraged if it takes some trial and error, because this exercise is complex.

To help you get started, look at Script 16-1, which draws a golden rectangle. Start by converting this script into a method that has the width of the rectangle as its parameter.

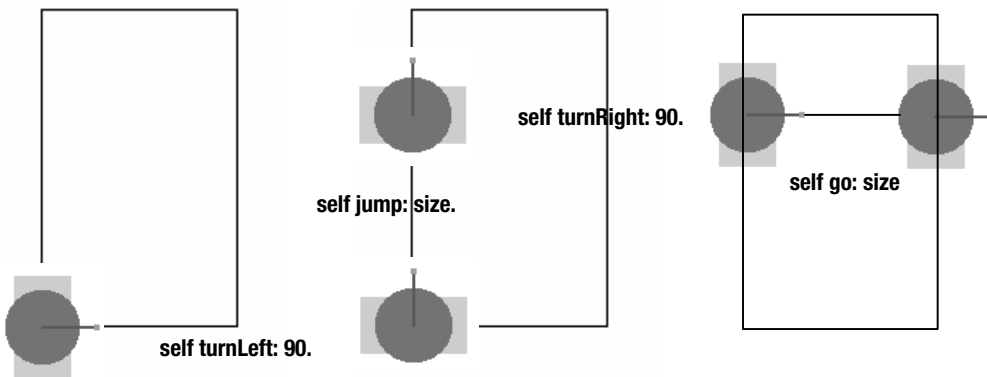
1. If the width of a golden rectangle is 1 and its length  $x$ , then when you cut a square out of the golden rectangle, the smaller golden rectangle has width and length 1. Since these two rectangles are similar, the ratios of their sides must be the same. Clearing denominators and solving the resulting quadratic equation reveals that the length  $x$  of the original rectangle must be equal to the golden ratio.

**Script 16-1.** *Pica draws a golden rectangle.*

```
| pica length width |
pica := Bot new.
width := 100.
length:= width * ((1 + 5 sqrt)/2).
2 timesRepeat:
 [pica go: width;
 turnLeft: 90;
 go: length;
 turnLeft: 90.]
```

## A One-Line-per-Rectangle Solution

Here is a solution based on the fact that you need to draw only the largest rectangle completely. After that, you can draw a single line in just the right place to create the next-smaller golden rectangle. This process is shown in Figure 16-4.



**Figure 16-4.** *Once a robot has drawn a golden rectangle, it moves into position and then draws a line that cuts the rectangle to form a square and a smaller golden rectangle.*

Thus to draw several golden rectangles you need to know how to (1) draw one entire golden rectangle, (2) draw a line that defines the square within the golden rectangle, and (3) compute the width of the next golden rectangle. Therefore, there are three tasks:

1. Define a method that given a width, knows how to compute the corresponding length and draws a golden rectangle, as I suggested you do in the previous section. Let's call this method `goldenRectangle: width`, where the parameter `width` represents the width of the golden rectangle.
2. Define a method that draws the line to delimit the square included inside the rectangle, as shown by the second rectangle in Figure 16-3. Let's call this method `dividingLine: sideLength`; it first moves the distance `sideLength` from the robot's starting point, and then draws a segment of length `sideLength` in the robot's initial direction. Note that these two distances are the same because they are two side lengths of a square.

3. Once the methods `goldenRectangle:` and `dividingLine:` have been defined, you will need a method that puts everything together: It should draw a golden rectangle, and then repeatedly draw the missing side of the square to form the next-smaller golden rectangle and compute the size of the next square, which is also the width of the next golden rectangle. You will also need to ensure that the robot starts at the right place and points in the right direction to draw the next square. Let's name this method `goldenRectangles: largestWidth atLevel: n`, where `largestWidth` represents the width of the first golden rectangle, and `n` indicates the number of squares that will be drawn (so you will end up with `n + 1` golden rectangles).

Now let us develop these three methods.

### The Method `goldenRectangle: width`

Method 16-1 is nothing special; given the width of a golden rectangle, it computes the length and then draws the rectangle.

**Method 16-1.** *Draw a golden rectangle given its width.*

```
goldenRectangle: width
 "Draw a rectangle whose length is the golden ratio times its width"

 | length |
 length := width * (1 + 5 sqrt / 2).
 2 timesRepeat:
 [self go: width ;
 turnLeft: 90 ;
 go: length ;
 turnLeft: 90]
```

The method `goldenRectangle:` starts drawing in the robot's current direction and draws the rectangle's width first. When it is finished, the robot is back at the starting point pointing in its original direction.

### The Method `dividingLine: sideLength`

The definition of the method `dividingLine:` assumes that the robot is at a corner and pointing along the width of the rectangle, with the rectangle's length at the robot's left. This scenario matches the situation at the end of the method `goldenRectangle:`. Figure 16-4 shows the action of `dividingLine:`, by which the robot turns ninety degrees to its left, walks along the rectangle's length, turns ninety degrees to the right, and then draws the missing side of the square to create a new golden rectangle.

In the situation of Figure 16-4, where the robot is at the bottom of a golden rectangle and pointing to the right (first panel), it ends up by drawing a horizontal segment of length `sideLength` at the distance `sideLength` above its starting position (third panel). This explanation is actually more complex than the definition of the method.

**Method 16-2.** *Draw a line that divides a golden rectangle into a square and a smaller golden rectangle.*

```
dividingLine: sidelength
 "Move the distance sidelength to the left of the starting position,
 and draw a segment of length sidelength in the initial direction."

 self turnLeft: 90 ;
 jump: sidelength;
 turnRight: 90 ;
 go: sidelength
```

### The Method `goldenRectangles: largestWidth atLevel: n`

This method, Method 16-3, is a bit more complex. Try to understand it yourself before reading the explanation that follows.

#### Method 16-3

```
goldenRectangles: largestWidth atLevel: n
 "Draw n + 1 golden rectangles; the largest of them has width largestWidth"
 (1) | currentRectangleWidth |
 (2) currentRectangleWidth := largestWidth.
 (3) self goldenRectangle: currentRectangleWidth.
 (4) n timesRepeat: [self dividingLine: currentRectangleWidth.
 (5) self turnLeft: 90.
 (6) currentRectangleWidth := currentRectangleWidth * ((1 + 5 sqrt) / 2) - 1]
```

This method produces a group of nested golden rectangles of which the first has `largestWidth` as its width. To create six golden rectangles, execute the expression `Bot new goldenRectangles: 100 atLevel: 5`. This is what happens:

- The variable `currentRectangleWidth` represents the width of the current rectangle. It also represents the size of the square segment to draw.
- Line (2) initializes the value of the variable `currentRectangleWidth` to the width of the first golden rectangle. This width is given as the argument of the method. This is the width of the first golden rectangle.
- Line (3) draws the first golden rectangle, which serves as the basis for all the others.
- Line (4) defines a loop that repeats the same actions `n` times.
  - Line (4) first draws the missing part of the square using the method `dividingLine:.` The size of this square is the width of the current golden rectangle. Therefore, the method uses the variable `currentRectangleWidth` as its argument. Note that using the variable `largestWidth` will not work for the smaller rectangles, because its value does not change when the loop is repeated.



- Line (5) positions the robot so that the next segment can be drawn. Our starting assumption was that the rectangle is always drawn with its width in the direction the robot is pointing. Therefore, the expression `self turnLeft: 90` points the robot along the width of the new rectangle (for example, after the last drawing of Figure 16-4, the robot will turn from east to north).
- The last line, line (6), comes from the observation that the length of a nested golden rectangle is in fact the width of the previous one. Therefore, the width of a new rectangle is the difference between the length and width of the previous rectangle. Expressed as some simple mathematical expressions, we get

$$\text{newWidth} = \text{currentRectangleLength} - \text{currentRectangleWidth}$$

From the formula for the golden ratio, we have

$$\text{currentRectangleLength} = \frac{1 + \sqrt{5}}{2} \times \text{currentRectangleWidth}$$

Therefore,

$$\text{newWidth} = \text{currentRectangleWidth} \times \left( \frac{1 + \sqrt{5}}{2} - 1 \right)$$

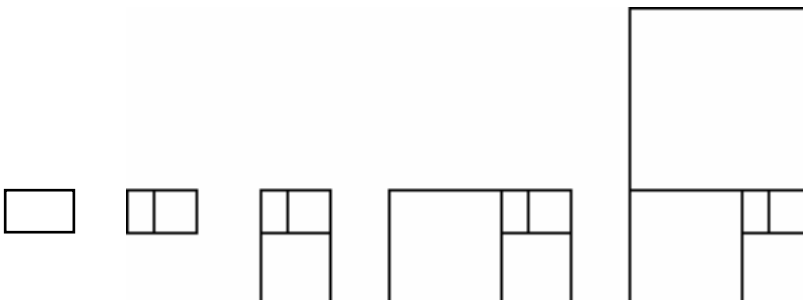
In Smalltalk, we write this as `currentRectangleWidth := currentRectangleWidth * ((1 + 5 sqrt) / 2) - 1`.

The individual problems of drawing a golden rectangle and drawing a dividing line are fairly simple. The composition of the solutions to these two problems to obtain the desired solution requires paying attention to the assumptions of each method, that is, the context for which it was defined.

You might want to try to find another solution to this problem. Then you can use the same technique with Experiment 16-10.

### Experiment 16-10 (Increasing Golden Rectangles)

You have seen a method for drawing a set of decreasing, or *nested*, golden rectangles. Now write a method to create a set of increasing, or *nesting*, golden rectangles. The figures below show some possible steps.



## Tiling

Now I would like you to try to reproduce some pictures using the method `square:.` These experiments are important for developing an understanding of how to compose methods and reuse the abstractions they represent. The idea promoted by these exercises is key to understanding how complex programs are built.

Here are some hints: First of all, there are several ways to approach the problems. In general, try to see how you can simplify the problem. If you look at the three figures associated with the next three experiments, you might imagine that it would be helpful if you had a method called, say, `lineOfSquares:` that could draw a row of an arbitrary number of squares. Assuming that you had such a method, sketch a solution to each problem. Then implement the method and see whether it helped you. Decomposing a problem is difficult, and the only way to learn is to invent and try something, evaluate the result, improve your invention, and try again.

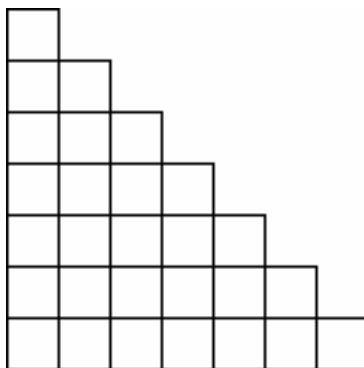
Alternatively, you could also use nested loops like these:

```
n timesRepeat: [
m timesRepeat: [...]]
```

I also suggest that you define your methods so they put the robot back in its starting position at the end of the method. This makes it easier to compose the methods. Experiment with all these approaches.

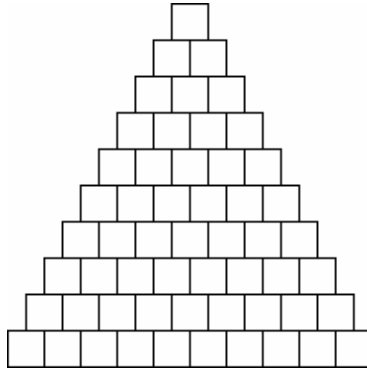
### Experiment 16-11 (A Rectangular Pyramid)

Using the method `square:.`, which draws one square of a given size, `lineOfSquares:.`, which draws a row of squares, and any other methods that you define, draw the pyramid shown below.



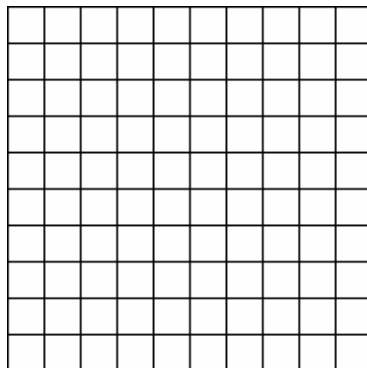
### Experiment 16-12 (A Triangular Pyramid)

Using the method `square:`, which draws one square of a given size, `lineOfSquares:`, which draws a row of squares, and any other methods that you define, draw the pyramid shown below.



### Experiment 16-13 (Checkerboard Squares)

Using the method `square:`, which draws one square of a given size, `lineOfSquares:`, which draws a row of squares, and any other methods that you define, draw the pyramid shown below.



## Summary

- To solve a big problem, decompose it into smaller problems; then compose the solutions of the smaller problems to build the solution to the large one.
- Pay attention to the context in which the small problems are solved, because you will need to make sure that their assumptions are valid when you compose them.



# Strings, and Tools for Understanding Programs

**I**n this chapter I present an important concept: the notion of strings. A string is a sequence of characters that can be used to represent words or sentences. One important use of strings is to communicate with users. In subsequent chapters, you will learn to use strings as a tool for understanding more complex program structures such as conditions and conditional loops. In this chapter I will present only the most important aspects of strings, providing the basic information that you will then use in subsequent chapters. I will also show you how to use strings to help you understand how programs are executed. I recommend that you also try to use the debugger for help in understanding the experiments that are proposed in this chapter.

## Strings

Strings are used to represent information and present it to the user. Strings are delimited by single quotes ('This is a string'), and as you can see, a string can contain white-space characters. For example, the string 'squeak is cool' represents a sequence of fourteen characters: s q u e a . . . Note that a space is also a character. A string can contain any number of characters, including zero. Thus '' is an empty string, 'a' is a string with only the character a, and ' ' is a string whose only character is a space.

---

**Important!** A string is a sequence of characters delimited by single quotes. A string represents textual information such as words or sentences. Strings can be used to display information on the screen.

---

Selecting a string and printing it (menu **print it**) prints that string. Several methods are defined on strings, but the most important one in the context of this book is the method `print`, whose name is the comma character. When the message `print` is sent to a string as receiver with one string as argument, the method `print` returns the concatenation of the two strings. That is, it returns a single string whose characters are those of the first string followed by those of the second.

```
'squeak' "the value of a string is itself"
-Printing the returned value: 'squeak'
```

```
'a' "a string can be composed of only one character"
```

```
'' "an empty string"
```

```
'squeak' , 'is cool' "concatenating two strings"
-Printing the returned value: 'squeakis cool'
```

```
'', 'squeak', ' ', 'is cool' "concatenating multiple strings"
-Printing the returned value: 'squeak is cool'
```

The method `copyReplaceAll` allows you to modify a string by replacing all occurrences of a particular substring with a different string. In the example below, 'not' is replaced with 'really':

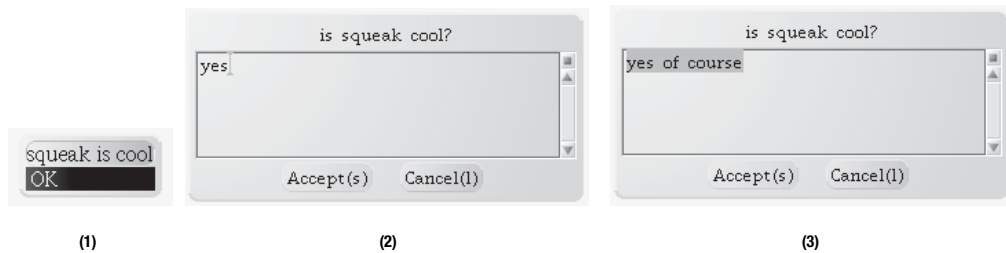
```
'Squeak is not cool' copyReplaceAll: 'not' with: 'really'
-Printing the returned value: 'Squeak is really cool'
```

## Communicating with the User

Squeak offers some tools to display information on the screen and to request information from the user. The class `PopupMenu` allows you to bring up a menu and display some information for the user using strings. For example, the expression `PopupMenu inform: 'squeak is cool'` causes a small window to pop up that displays the string 'squeak is cool' and then waits until the user presses the **ok** button. The class `FillInTheBlank` allows you to request

input from the user. For example, the expression `FillInTheBlank request: 'is squeak cool?'` brings up a dialog box with an input field and waits for the user to fill in the input field and press the **accept** button or the **cancel** button. The result of this expression is a string that represents what was typed by the user. You can also specify a default user input using the message `request:initialAnswer:` as shown in the following script, and in Figure 17-1.

- (1) `PopUpMenu inform: 'squeak is cool'`
- (2) `FillInTheBlank request: 'is squeak cool?'`  
*-Printing the returned value: 'yes'*
- (3) `FillInTheBlank request: 'is squeak cool?' initialAnswer: 'yes of course'`  
*-Printing the returned value: 'yes of course'*



**Figure 17-1.** Pop-up menu and fill in the blanks

## Strings and Characters

A string is composed of characters. While a string is enclosed in single quotes to show that it is a string, individual characters are prefixed by a dollar sign \$ to show that they are characters. For example, `$a` represents the character representing the letter “a”. Note that while individual characters are prefixed by the dollar sign \$, when you edit a string you simply type the characters without the dollar sign.

There are several methods for accessing the individual characters of a string. For example, the methods `first`, `second`, and `third` return the first, second, and third characters of a string. The method `size` returns the number of characters in a string, while the method `at: aNumber` returns the character located at the specified position in the string. You can replace the character at a specified position with another character using `at: aNumber put: aCharacter`. The method `copyUpTo: aCharacter` returns the beginning of a string up to the first character that matches `aCharacter`. Here are some examples:

```
'squeak is cool' first
-Printing the returned value: $s
```

```
'squeak is cool' size
-Printing the returned value: 14
```

```
'squeak' at: 5
-Printing the returned value: $a

'squeak is cool' at: 11 put: $f
-Printing the returned value: 'squeak is fool'

'squeakiscool' copyUpTo: $i
-Printing the returned value: 'squeak'

'squeak is cool' copyUpTo: Character space
-Printing the returned value: 'squeak'
```

To create a character that does not have a graphical representation, such as the space, tab, or carriage return character, you can send a message to the class `Character`. The messages `Character space`, `Character tab`, and `Character cr` return respectively the space, tab, and carriage return characters.

Script 17-1 shows how to insert a carriage return into a string. Note that the method `at:put:` does not return the modified string, but the character that was inserted. This is an example where the *effect* of the message and its *result* are clearly different. Printing the result of the message `'squeak is cool' at: 7 put: Character cr` does not illustrate the effect of the method. Therefore, we print instead the modified string. To review how to print results of a message send on the screen, see Chapter 5, “Pica’s Environment.”

**Script 17-1.** *Inserting a carriage return into a string*

```
|myString|
myString:= 'squeak is cool'.
myString at: 7 put: Character cr.
myString
-Printing the returned value: 'squeak
is cool'
```

A character can also be converted into a string by sending it the message `asString`. In Script 17-2, three strings are concatenated together. The middle one is the string `'a'` created by the message `send $a asString`.

**Script 17-2.** *A character is converted into a string, which is then concatenated with other strings.*

```
'sque', $a asString, 'k'
-Printing the returned value: 'squeak'
```

## Strings and Numbers

A string can represent a number. For example, the *string* '10' is a textual representation of the *number* 10. However, a string is not a number. A string does not know how to perform any mathematical operation, and a number does not know how to behave as a string. For example, we cannot concatenate two numbers or add two strings. However, a number knows how to produce a string that *represents* it using the method `asString`. In addition, a string knows how to convert a representation of a number into a number using the method `asNumber`.

Thus there is a difference between the number 10 and the string '10'. The number 10 represents the mathematical quantity 10, while the string '10' represents the textual representation of the number 10 that consists of the two characters 1 and 0. The string '10' is composed of the two characters: \$1 and \$0, and the string '12' is composed of the two characters \$1 and \$2. Here are some illustrations of operations with strings and numbers:

```
10 , 12
-> error! a number does not know the message ,
```

```
'10', '12'
-Printing the returned value: '1012'
```

```
10 asString
-Printing the returned value: '10'
```

```
10 asString , 12 asString
-Printing the returned value: '1012'
```

```
'10' asNumber
-Printing the returned value: 10
```

---

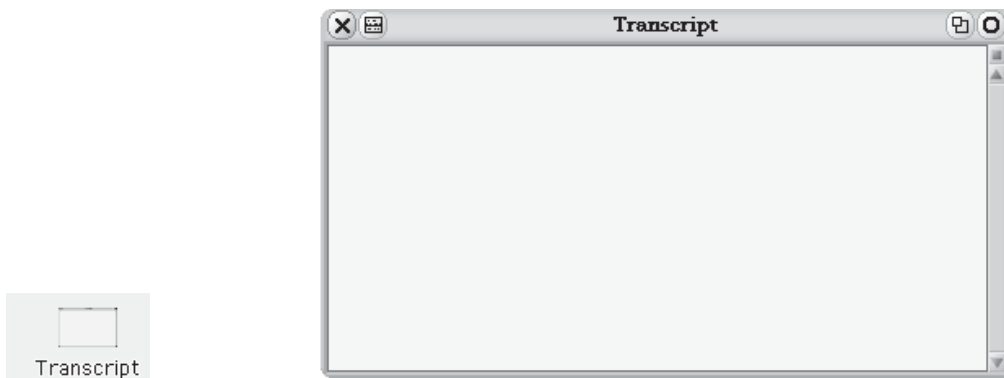
**Note** A string can represent a number, but such a string is not a number. For example, the string '79' is composed of the two characters: \$7 and \$9. To obtain the string representing a number, send the message `asString` to it.

---



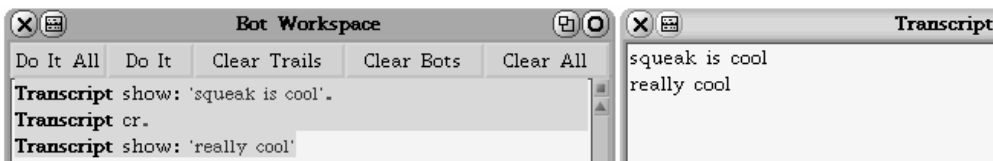
## Using the Transcript

Squeak offers several powerful tools for understanding program execution, such as the debugger (see Chapter 15). Another tool is called Transcript, with which you can display information in the form of strings. To open a transcript window, drag the thumbnail that is available in one of the flaps onto the desktop, or you can choose the **transcript** item of the **open...** menu. This opens a window, as shown in Figure 17-2.



**Figure 17-2.** To open a transcript window, drag and drop the thumbnail that you can find in a flap.

There are two main messages for displaying information in a transcript window: `show:` and `cr`. The message `show: aString` displays the given string in the window, and the message `cr` inserts a new line. See Figure 17-3.



**Figure 17-3.** Writing to the transcript

Script 17-3 gives some examples of displaying information in a transcript window.

### Script 17-3. Displaying information in a transcript window

```
Transcript show: 'squeak is cool'.
Transcript cr.
Transcript show: 'really cool'
```

Note that a transcript window can display only strings. And so if you want to display a number, you have to obtain a string representing it using the method `asString`. This is illustrated in Script 17-4.

**Script 17-4.** *A number is converted into a string before it is displayed.*

```
Transcript show: '21 + 21 is: ', 42 asString ; cr
```

## Generating and Understanding a Trace

Now I would like to show you how you can use Transcript to generate a trace of a program. A trace is a collection of indications of what is going on that is generated by a program. For example, you might want to track a robot's movements in a script by having the script print 'I am turning right' every time the robot makes a right turn. To generate a trace, you simply introduce one or more expressions into your script that do not change the original execution of the program but, for example, display information in a transcript window. Let us begin with Script 17-5, which draws a staircase with treads of increasing length.

**Script 17-5.** *A staircase with treads of increasing length*

```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
10 timesRepeat:
 [pica go: treadLength.
 pica turnLeft: 90.
 pica go: 5.
 pica turnRight: 90.
 treadLength:= treadLength + 10]
```

The first simple trace that we can generate tells us when the program is about to enter the `timesRepeat:` loop and when it has exited the loop. This is shown in Script 17-6.

**Script 17-6.** *The staircase with a simple trace*

```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
Transcript show: 'Before the loop' ; cr.
10 timesRepeat:
 [pica go: treadLength.
 pica turnLeft: 90.
 pica go: 5.
 pica turnRight: 90.
 treadLength:= treadLength + 10].
Transcript show: 'After the loop' ; cr.
```

### Experiment 17-1 (Putting a Trace inside the Loop)

Modify Script 17-6 by introducing the expression `Transcript show: 'inside the loop' ; cr.` inside the loop. Your transcript should now print 'inside the loop' ten times, one for each pass through the loop. You can also insert the expression `self halt` to allow you to open the debugger. But watch out, or you will get ten debuggers!

Now I would like to use the same technique to generate a more sophisticated trace. For example, it would be nice to see how the value of the variable `treadLength` evolves while the program is executed. Script 17-7 contains a new expression that prints the value of the variable `treadLength` at the beginning of the loop each time it is executed. The results are shown in Figure 17-4.

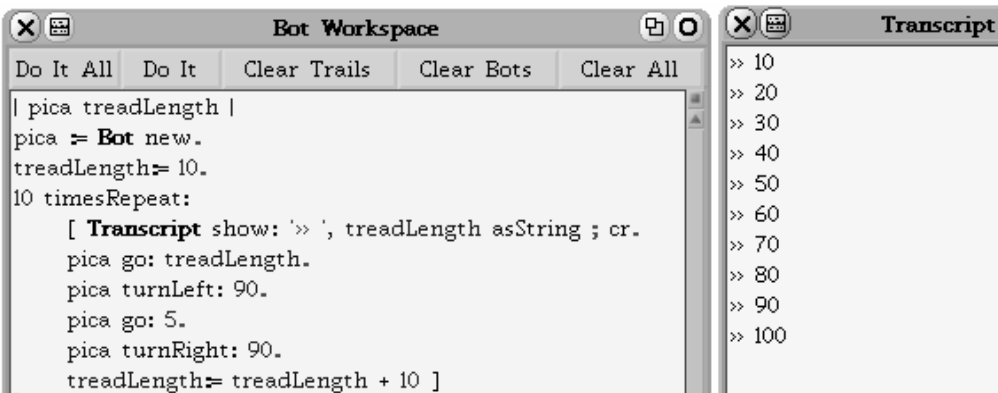


Figure 17-4. Adding a trace to a script

#### Script 17-7. The staircase with a more sophisticated trace

```

| pica treadLength |
pica := Bot new.
treadLength:= 10.
10 timesRepeat:
 [Transcript show: '>> ', treadLength asString ; cr.
 pica go: treadLength.
 pica turnLeft: 90.
 pica go: 5.
 pica turnRight: 90.
 treadLength:= treadLength + 10]

```

Adding a trace after an assignment is often useful, since it reveals some key behavior of a program. For example, in Script 17-8, the expression `Transcript show: 'After := ', treadLength asString ;cr.` has been added after the assignment statement that is the last expression of the loop. The trace, which is shown following the script, prints the value of the variable `treadLength` at the beginning and the end of the loop. These two values should be the same, and indeed, they are, as the trace shows.

**Script 17-8.** *The staircase with a trace after an assignment*

```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
10 timesRepeat:
 [Transcript show: 'treadLength: ', treadLength asString ; cr.
 pica go: treadLength.
 pica turnLeft: 90.
 pica go: 5.
 pica turnRight: 90.
 treadLength := treadLength + 10.
 Transcript show: ' treadLength after := ', treadLength asString ; cr.]
```

And here is the trace:

```
treadLength: 10
treadLength after := 20
treadLength: 20
treadLength after := 30
treadLength: 30
treadLength after := 40
treadLength: 40
treadLength after := 50
treadLength: 50
treadLength after := 60
treadLength: 60
treadLength after := 60
treadLength: 70
treadLength after := 70
treadLength: 80
treadLength after := 90
treadLength: 90
treadLength after := 100
treadLength: 100
treadLength after := 110
```

## Summary

- A string is a sequence of characters delimited by single quotes: 'This is a string'. A string represents textual information such as words or sentences and can be used to display information on the screen. For example, 'squeak is cool' is a string of 14 characters.
- A character is one letter prefixed by the dollar sign \$. Thus \$a represents the character a.
- A string can represent a number, but such a string is not a number. For example, the string '79' is composed of the two characters: \$7 and \$9. To obtain the string representing a number, send the message asString to it.
- A Transcript window is a small window used to display messages. The message show: aString displays the value of the argument aString, which must be a string, in the transcript window. The message cr adds a new line in the transcript window.

## PART 4



# Conditionals

**U**p until now, all of your programs have executed all of their expressions. You have had no way to express that certain parts of a program should be executed only when certain conditions are met. In this part, I present conditional expressions, which solve this problem. This part also introduces the notion of *references* in two-dimensional space as well as some other robot behavior. Finally, I will show you how to use a robot to simulate the behavior of simple animals.



# Conditions

Up to this point, the programs you have defined execute *all* the expressions they contain, one after the other. You had no way of saying that certain expressions should be executed only when certain *conditions* were met. This chapter and the next one introduce an important programming concept: the notion of *conditional* execution, that is, the execution of a certain piece of code only when a specified condition holds. Formally, a *condition* is an expression that can be either true or false.

This chapter starts by defining a simple problem that shows the need for conditional execution. Then I will show you that a conditional expression is composed of a *condition* and a *conditional message* that takes one or more arguments whose execution depends on the value of the condition. The arguments of a conditional message are called *conditional blocks*, which are sequences of expressions enclosed in square brackets [ ].

## A Robot's True Colors

Suppose you want to change the color of a robot depending on its distance from the center of the screen. If a robot is less than 200 pixels from the center, it should be red. Otherwise, it should be green. This problem requires *conditional* execution. Depending on a condition—the robot's location—its color should change.

The method `distanceDetector`, which appears as Method 18-1, shows a possible solution, and Script 18-1 shows how the method `distanceDetector` can be used.

**Script 18-1.** *Pica changes color according to the method `distanceDetector`.*

```
| pica |
pica := Bot new.
pica jump: 20.
pica distanceDetector.
pica jump: 200.
pica distanceDetector.
```

**Method 18-1.** *A distance detector method*

**distanceDetector**

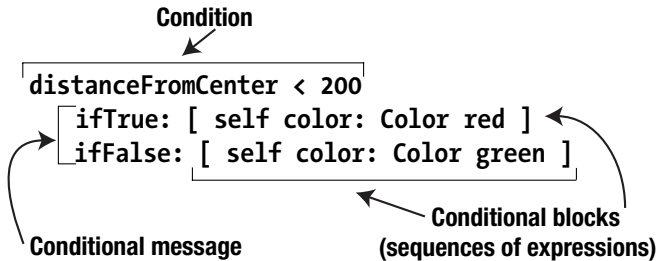
```
| distanceFromCenter |
distanceFromCenter := self distanceFrom: World center.
distanceFromCenter < 200
 ifTrue: [self color: Color red]
 ifFalse: [self color: Color green]
```

Let's analyze what happens when the expression `pica distanceDetector` is executed. First, the expression `self distanceFrom: World center` computes the distance from the receiver to the center of the screen. This distance is stored in the variable `distanceFromCenter`.

Then the expression `distanceFromCenter < 200 ifTrue: [ self color: Color red ] ifFalse: [ self color: Color green ]` is executed as follows: *if* the distance from the center is less than 200, the color of the receiver is changed to red; otherwise, it is changed to green. This expression is a *conditional* expression. It is spread over three lines in the method definition, but it is all one expression, which you can tell because there is no period between any of its parts.

Once again, a conditional expression is composed of two parts: a *condition* and a *conditional message* whose arguments are *conditional blocks*. The expression `distanceFromCenter < 200` is a condition, the expression `ifTrue: [ self color: Color red ] ifFalse: [ self color: Color green ]` is a conditional message, and `[ self color: Color red ]` and `[ self color: Color green ]` are conditional blocks, as shown in Figure 18-1.





**Figure 18-1.** A conditional expression is composed of a condition and a conditional message, whose arguments are conditional blocks.

The method `ifTrue:iffalse:` executes one condition, here `distanceFromCenter < 200`, and depending on its value, executes one of the conditional blocks and skips the other. The keyword `ifTrue:` indicates that the conditional block `[ self color: Color red ]` is executed only if the condition is true. Similarly, the keyword `iffalse:` indicates that the conditional block `[ self color: Color green ]` is executed only if the condition is false. Conditional blocks are also called condition *branches*. Imagine that you are tracing up the trunk of a tree in a picture with your finger, and each time you encounter a branch, you have to choose which path to follow, and you can follow only one path at a time. The term *branch* refers to such a situation. It represents the fact that execution of the program has to choose between branches and execute only one branch, while skipping over the other one.

You have now seen two different kinds of expressions: one kind, such as `self distanceFrom: World center`, are always executed when they are encountered in a program, while others, those inside conditional blocks, are executed only when their associated condition is true or false, as the case may be.

A conditional block is not limited to a single expression. Rather, it can contain a sequence of expressions, as I will present in the next chapter. Thus the method `ifTrue:iffalse:` defines two conditional blocks, each containing a *sequence* of expressions. (As you will see in the next section, you can even have a sequence of no expressions.)

Note finally that the method `ifTrue:iffalse:` is a *single* method with two arguments, one for the true case and one for the false case. Therefore, you must not put a period after the right bracket `]` terminating the `ifTrue:` block, since that would break up the conditional statement by ending it too soon, causing an error.

## Adding a Trace to See What Is Going On

To understand how conditional expressions are executed, you should experiment with sending messages to Transcript to generate a trace, as presented in Chapter 17. You can also use the debugger by inserting the expression `self halt`, as shown in Chapter 15. If you want to know whether a particular branch is being executed, introduce expressions in the branch. For example, you could place the expression `Transcript show: 'now in the true branch'; cr` in the `ifTrue:` branch. Method 18-2 presents one way to generate such a trace in the context of the `distanceDetector` method. Depending on the position of the robot, different traces will be produced. Try to guess what they will be before executing Script 18-1. Do not hesitate to add and modify such traces in all the scripts that you find difficult to understand.

**Method 18-2.** *The distance detector method with a trace*

```
distanceDetector
| distanceFromCenter |
distanceFromCenter := self distanceFrom: World center.
Transcript show: 'always'; cr.
distanceFromCenter < 200
 ifTrue: [self color: Color red.
 Transcript show: 'red' ; cr]
 ifFalse: [self color: Color green
 Transcript show: 'green' ; cr]
```

## The Value Returned by a Method

When I showed you how to define methods in Chapter 12, I explained that executing a method not only evaluates the messages it contains but also *returns* a value. Up until now, we have not particularly emphasized the results returned by methods. Now, however, we are going to be especially concerned with the value returned by the condition that begins a conditional expression. For example, the condition `distanceFromCenter < 200` returns a value (true or false) that tells whether the distance to the center of the screen is or is not less than 200 (you will learn more about true and false values in Chapter 20).

Other expressions that appear in Method 18-1 return values of interest. For instance, not only does the expression `self distanceFrom: World center` compute the distance of the receiver from the center of the screen, but it also *returns* that distance. Then the condition `distanceFromCenter < 200` uses this distance to decide which branch of the conditional expression should be executed.

---

**Important!** A conditional expression is composed of a *condition* and a *conditional message* that takes one or more arguments and whose execution depends on the value of the condition. The arguments of a conditional message are called *conditional blocks*, which are sequences of expressions enclosed in square brackets [ ]:

```
aCondition
 ifTrue: [expressionsIfConditionIsTrue]
 ifFalse: [expressionsIfConditionIsFalse]
```

---

## Conditional Expressions with Only One Branch

Sometimes, you need to perform an action when a certain condition is true, but if the condition is false, then you want to do nothing (or vice versa). To take an example from real life, if you are carrying a closed umbrella, then if it is starting to rain, you want to open it, but if it is not starting to rain, you don't want to do anything special (do nothing). For an example from Squeak, the method `redWhenCloseToCenter` (Method 18-3) changes the color of the receiver to red when it is at a distance smaller than 200 pixels from the screen center, but if the distance is not smaller than 200 pixels, then the robot's color remains unchanged.

**Method 18-3.** *If you want to do nothing if the condition is false, you can use an empty conditional block with `ifTrue:ifFalse:`.*

### `redWhenCloseToCenter`

```
| distanceFromCenter |
distanceFromCenter := self distanceFrom: World center.
distanceFromCenter < 200
 ifTrue: [self color: Color red]
 ifFalse: []
```

With the method `ifTrue:ifFalse:`, you leave the second branch empty, resulting in a conditional block with no expressions: [ ]. However, Smalltalk provides two other methods, `ifTrue:and` and `ifFalse:`, to express these kinds of conditional expressions. The method `ifTrue:` executes its conditional block when its condition is true. Using the method `ifTrue:`, I can rewrite Method 18-3 more conveniently as Method 18-4.

**Method 18-4.** *With the method `ifTrue:`, you no longer require an empty conditional block.*

### `redWhenCloseToCenter`

```
| distanceFromCenter |
distanceFromCenter := self distanceFrom: World center.
distanceFromCenter < 200
 ifTrue: [self color: Color red]
```

The method `ifFalse:` is completely analogous to the method `ifTrue:`, executing its conditional block when its condition is *false*. Each of the methods `ifTrue:` and `ifFalse:` executes a condition, and then, depending on the value returned by the condition, the method either executes or skips the conditional block.

---

■ **Important!** The method `ifTrue:` executes its conditional block when its condition is true. The method `ifFalse:` executes its conditional block when its condition is false:

*aCondition*

```
ifTrue: [expressionsIfConditionIsTrue]
```

*aCondition*

```
ifFalse: [expressionsIfConditionIsFalse]
```

---

## Choose the Right Conditional Method

Using `ifTrue:ifFalse:` is not the same as using `ifTrue:` followed by `ifFalse:`. With `ifTrue:ifFalse:`, the condition that precedes it is executed only once, but if you want to use `ifTrue:` followed by `ifFalse:`, then you need a condition before each message, and both of them will be executed, even if they are identical. Potentially, this could lead to unintended consequences if the conditional block of the first conditional expression (`ifTrue:`) modifies what is tested by the condition of the second conditional expression (`ifFalse:`). In such a case, using `ifTrue:` followed by `ifFalse:` would not be equivalent to using `ifTrue:ifFalse:`.

## Nesting Conditional Expressions

A conditional expression can contain any other expressions inside its conditional blocks, and in particular, these blocks can contain other conditional expressions. This is what I will explain next. There is nothing conceptually new about this, but it is a common and useful practice, and that is why I want to show it to you.

### Robot Coloring with Three Colors

Let's modify the previous problem. If a robot is less than 200 pixels from the center of the screen, then it should be colored red; if it is between 200 and 300 pixels away, it should be colored yellow; and if it is a distance greater than 300, it should be colored green.

In this problem, different parts of the method should be executed under different circumstances. That is, there are three different ranges of distances, and the robot's color should change to yellow, green, or red depending on which range it is in. A possible solution to our problem is shown by Method 18-5. What I have done is to break the problem into two pieces: what to do if the robot's distance to the center is greater than 300 pixels, and what to do if it is not. Then I break the "if it is not" part again into two subproblems: what to do if the distance to the center is less than 200, and what to do if it is not.

**Method 18-5.** *Color the robot one of three colors depending on its distance from the center.*

#### **setThreeColor**

```
| distance |
distance := self distanceFrom: World center.
distance > 300
 ifTrue: [self color: Color green]
 ifFalse: [distance < 200
 ifTrue: [self color: Color red]
 ifFalse: [self color: Color yellow]]
```

Method 18-6 contains the exact same code with typographical emphasis added to show the conditional expressions. There are two different conditional expressions. The first one (conditional expression 1 below) is shown in italics, and the second (conditional expression 2) in bold. The second conditional expression is executed only if the condition of the first conditional expression is false. That is, if the distance is less than or equal to 300, then conditional expression 2 is executed. If condition 2 is executed, then its condition `distance < 200` is executed, and depending on its value, the `ifTrue:` or `ifFalse:` branch is executed. Here are the two conditions:

#### **Conditional Expression 1:**

```
distance > 300
 ifTrue: [self color: Color green]
 ifFalse: ["execute conditional expression 2"]
```

#### **Conditional Expression 2:**

```
distance < 200
 ifTrue: [self color: Color red]
 ifFalse: [self color: Color yellow]
```

And here is Method 18-6.

**Method 18-6.** *This is Method 18-5 with emphasis added.*

**setThreeColor**

```
| distance |
distance := self distanceFrom: World center.
distance > 300
 ifTrue: [self color: Color green]
 ifFalse: [distance < 200
 ifTrue: [self color: Color red]
 ifFalse: [self color: Color yellow]]
```

If you are having trouble identifying which conditional block will be executed, choose a few particular values for the distance (such as 150, 250, and 350). Trace through the method and underline the part of each method that will be executed. Following each step carefully will show you that only certain branches are executed. You can also introduce different traces to show how the different conditions are executed, or you can use the debugger to go step by step through the method.

## Learning from Your Mistakes

Since everyone makes mistakes, studying your programming errors is an excellent way to learn and understand a concept from another perspective. I have decided to define a method `coloredTurn: anAngle` that changes the color of a robot according to the direction in which it is pointing. When the robot points to the north, I would like it to turn blue to represent cold. And I would like the robot to turn red when it points to the south, and otherwise, it should be green. My first attempt, which was not entirely successful, at defining this method is presented as Method 18-7.

**Method 18-7.** *The robot's color depends on its direction: first attempt (with a bug).*

**coloredTurn: anAngle**

```
"change the color of the robot so that it is blue when
pointing north, red when pointing south, and green otherwise"

self turn: anAngle.
self direction = 90
 ifTrue: [self color: Color blue].
self direction = -90
 ifTrue: [self color: Color red]
 ifFalse: [self color: Color green]
```

The definition of Method 18-7 is not correct. Try to understand why before reading any further. Follow through the method definition to see that when a robot is pointing north, it will first get colored blue, as it should, but then it will become colored green, which it should not. It should stay blue after being colored blue. Script 18-2 illustrates the bug in the method.

**Script 18-2.** *There is a bug in the method!*

```
| pica |
pica := Bot new.
pica coloredTurn: -90.
pica color
-Printing the returned value: Color red "ok"
pica coloredTurn: 90.
pica color
-Printing the returned value: Color green. "ok"
pica coloredTurn: 90.
pica color
-Printing the returned value: Color green "wrong"
```

*What went wrong?* Execute Method 18-7 mentally to identify the bug. The problem is that when the robot is pointing north, the condition `self direction = 90` is true, and so its associated block is executed, coloring the robot blue. We should be done. But then the method continues and executes the conditional block of the second conditional expression, and since the direction of the robot is not south, the robot's color is changed to green. The following commented version of the code illustrates this.

```
pica coloredTurn: 90.

self direction = 90 "is true"
 ifTrue: [self color: Color blue] "so the true conditional message is
 executed; the robot becomes blue
 and evaluates the next condition"

self direction = -90 "is false"
 ifFalse: [self color: Color green] "so the false conditional
 message is executed. Bug!"
```

*How to fix it?* In tracing through the method, you saw that when the robot is pointing north, the second conditional should not be executed. It should be executed only if the first condition is false. Therefore, you can use nested conditional expressions. Correct code is shown in Method 18-8.

**Method 18-8.** *The robot's color depends on its direction: correct version.*

**coloredTurn: anAngle**

"change the color of the robot so that it is blue when pointing north, red when pointing south, and green otherwise"

```
self turn: anAngle.
self direction = 90
 ifTrue: [self color: Color blue]
 ifFalse: [self direction = 90
 ifTrue: [self color: Color red]
 ifFalse: [self color: Color green]]
```

## Interpreting a Tiny Language

In theoretical biology, researchers have developed systems called Lindemeyer systems for studying the growth of plants. Lindemeyer systems are based on robot graphics similar to the robots that you have been experimenting with. Lindemeyer robots understand a tiny language composed of characters such as \$g and \$t. A robot's action is associated with each of these characters. For example, the character \$g (for "go") is associated with moving forward, while \$t (for "turn") is associated with turning through an angle of 45 degrees. A Lindemeyer system generates a sequence of characters. These characters are then interpreted by a robot, and the sequence of actions that it takes produces a picture.

Let us define a method interpret: aCharacter that makes a robot move forward if the character is \$g and turn counterclockwise 45 degrees if the character is \$t. Script 18-3 illustrates how this method is used, and the method is defined in Method 18-9.

**Script 18-3.** *Using the method interpret: aCharacter.*

```
| pica |
pica := Bot new.
4 timesRepeat:
 [pica
 interpret: $g;
 interpret: $t;
 interpret: $g;
 interpret: $g;
 interpret: $t;
 interpret: $g]
```



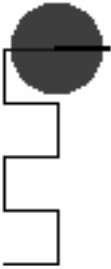
**Method 18-9.** *Interpreting a character***interpret: aCharacter**

```

aCharacter = $g
 ifTrue: [self go: 20]
 ifFalse: [aCharacter = $t
 ifTrue: [self turn: 45]]

```

Try to write a script to reproduce Figure 18-2. Since a string is a sequence of characters, it encodes a picture generated by a Lindenmeyer robot. Although it uses some methods that I have not yet explained (they will be explained in the sequel to this book), try out Script 18-4, which repeatedly sends the message `interpret:` to the robot with each character of the string. Then experiment further by changing the string `'gTTgTgTTTTgTTTTgTgTgTTTTgTTTTg'` to something else to create your own pictures. You will build a complete Lindenmeyer system in the sequel to this book.



**Figure 18-2.** *A picture generated using the method `interpret:` with the sequence of characters `'gTTgTgTTTTgTTTTgTgTgTTTTgTTTTg'`.*

**Script 18-4.** *Using `interpret:` in a loop*

```

| pica |
pica := Bot new.
'gTTgTgTTTTgTTTTgTgTgTTTTgTTTTg'
 do: [:aChar | pica interpret: aChar]

```

**Further Experiments**

Enhance the method `interpret: aCharacter` so that either `$g` or `$G` will make the robot go forward, and either `$t` or `$T` will make it turn 45 degrees counterclockwise. Also use the character `$+` before `$t` or `$T` to make the robot turn 45 degrees counterclockwise, and similarly `$-` to make it turn 45 degrees clockwise.

## Summary

A conditional expression is composed of two parts: a *condition* and a *conditional message*, which contains one or more *conditional blocks*. Which conditional block or blocks are executed depends on the value of the condition. The following table shows some of the methods used with conditions:

| Method                                                                                                      | Description                                                                                                                              | Example                                                                                |
|-------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <i>aCondition</i><br>ifTrue: [ expressionsIfConditionIsFalse ]                                              | Execute <i>expressionsIfConditionIsTrue</i> only if aCondition is true.                                                                  | self direction = 90<br>ifTrue: [ self color: Color green ]                             |
| If a robot is pointing to the north, it turns green.                                                        |                                                                                                                                          |                                                                                        |
| <i>aCondition</i><br>ifFalse: [ expressionsIfConditionIsFalse ]                                             | Execute <i>expressionsIfConditionIsFalse</i> only if aCondition is false.                                                                | self direction = 90<br>ifFalse: [ Beeper beep ]                                        |
| The system beeps only when the robot is not pointing to the north.                                          |                                                                                                                                          |                                                                                        |
| <i>aCondition</i><br>ifTrue: [ expressionsIfConditionIsTrue ]<br>ifFalse: [ expressionsIfConditionIsFalse ] | Execute <i>expressionsIfConditionIsTrue</i> whether <i>aCondition</i> is true; otherwise, execute <i>expressionsIfConditionIsFalse</i> . | self direction = 90<br>ifTrue: [ self color: Color green ]<br>ifFalse: [ Beeper beep ] |



# Conditional Loops

**C**onditional expressions are a powerful tool for creating complex programs because they allow you to control the flow of execution, branching one way if a condition is true and branching the other way if it is false. However, conditions are not enough. Some programs need to combine loops and conditions into *conditional loops*, that is, loops that execute a block of expressions *while* a certain condition holds, stopping execution when the condition no longer holds. This chapter presents the conditional loops offered by Smalltalk and introduces some simple examples. Then, in Chapter 23, we will use conditional loops to simulate animal behavior.

## Conditional Loops

The idea behind conditional loops is that a block (a sequence of expressions) is repeated as long as a certain condition holds (or alternatively, as long as a certain condition doesn't hold). Smalltalk defines two messages, `whileTrue:` and `whileFalse:`, that allow you to define conditional loops.

What such loops accomplish is indicated by method name: `whileTrue:` executes its condition and executes the conditional block as long as (while) the condition is true. The method `whileFalse:` does the same thing, but executes the conditional block only while the condition is false.

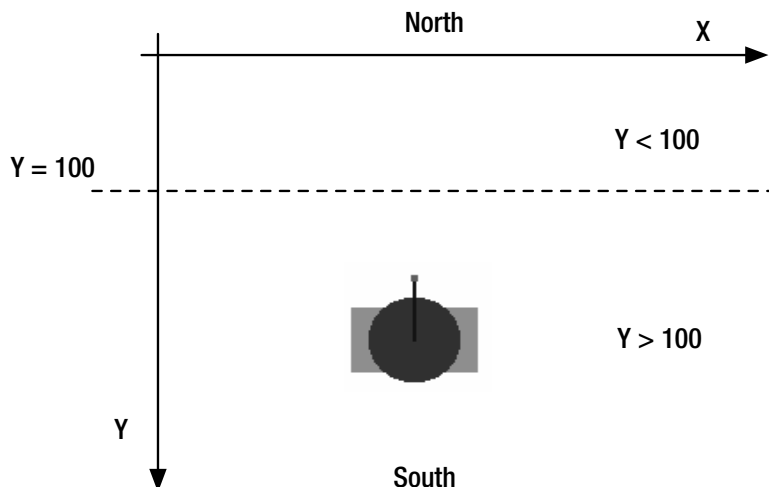
---

**Important!** `whileTrue:` and `whileFalse:` allow you to define conditional loops for which the conditional block is repeated while the condition holds (`whileTrue:`) or while the condition does not hold (`whileFalse:`).

---

### An Example

Let's take a simple example. Imagine that we have a robot somewhere down in the southern part of the screen where the  $y$ -coordinate is greater than 100 (in the Squeak environment, the positive  $y$ -axis runs from downward, from north to south), and we want the robot to move north until its  $y$ -coordinate is less than 100 pixels, as shown in Figure 19-1. A solution using a conditional loop is shown in Method 19-1, and it can be invoked as shown in Script 19-1.



**Figure 19-1.** The method `upTo100` moves the robot north while its  $y$ -coordinate is greater than 100.

**Script 19-1.** Invoking the method `v`

```
| pica |
pica := Bot new.
pica upTo100
```

**Method 19-1.** *Moving the robot north while its y-coordinate is greater than 100*

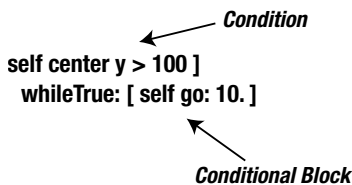
```
upTo100
```

```
"Turn the receiver north and move it ten pixels at a time
until its y-coordinate is less than 100. Then the receiver turns green."
```

```
self north.
[self center y > 100]
 whileTrue: [self go: 10].
self color: Color green.
```

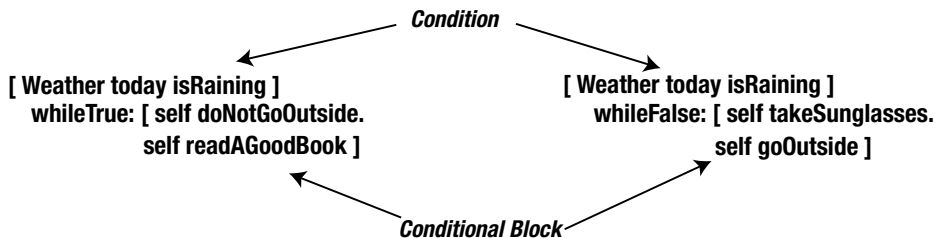
Let's look carefully at what happens when this method is executed.

1. The expression `self north` is not part of the conditional loop, so it is executed once.
2. The conditional loop is composed of a condition and a conditional message, as shown in Figures 19-2 and 19-3. The condition, which here is expressed as the block `[ self center y > 100 ]`, is executed (returning a true or false value).



**Figure 19-2.** A `whileTrue:` conditional loop is composed of a condition and a conditional message, which contains a conditional block (a sequence of expressions).

3. The method `whileTrue:` specifies the nature of the loop: If the result of the condition in step 2 is true, then the conditional block `[ self go: 10 ]` is executed, and after execution of the conditional block terminates, the method execution resumes back at step 2 (where the condition is executed again). On the other hand, if the condition is false, then execution continues at step 4.



**Figure 19-3.** The `whileTrue:` and `whileFalse:` conditional loops are composed of a condition and a conditional message. The conditional block for `whileFalse:` is executed as long as `whileFalse:`'s condition is false; `whileTrue:`'s conditional block is executed as long as `whileTrue:`'s condition is true.

4. At this step, the result of the condition [ self center y > 100 ] in step 2 was false, so the conditional block was not executed, and the loop stops. The program goes on to step 5.
5. Execution of the method resumes at the first expression after the conditional block. In this example, the expression self color: Color green gets executed, and the method terminates.

---

**Important!** A conditional loop is composed of a condition and a conditional message, which contains a conditional block (a sequence of expressions).

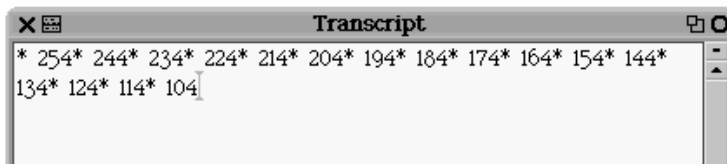
```
[condition] whileFalse:
 [conditional messages]
```

```
[condition] whileTrue:
 [conditional messages]
```

---

## Experiences with Traces

I suggest that you add traces in the method you just wrote and analyze the resulting trace. Also, use the debugger. Do not hesitate to modify the location of your trace in the method, because different placements of Transcript show: can result in different traces. For example, in Method 19-2, I introduced a trace at the beginning of the conditional block. I obtained the trace shown in Figure 19-4.



**Figure 19-4.** A trace of the execution of the method upTo100

**Method 19-2.** *The method upTo100 with a trace inside the loop*

**upTo100**

"Turn the receiver north and move it ten pixels at a time  
until its y-coordinate is less than 100. Then the receiver turns green."

```
self north.
[self center y > 100]
 whileTrue:
 [Transcript show: '* ' , self center y asString.
 self go: 10].
self color: Color green
```

As an alternative, you can introduce the line `Transcript show: '# ' , self center y asString ; cr.` after the expression `self go: 10`, as shown in Method 19-3, or even inside the condition block, before the first line, as shown in Method 19-4.

**Method 19-3.** *Placing transcripts in the method upTo100*

**upTo100**

"Turn the receiver north and move it ten pixels at a time  
until its y-coordinate is less than 100. Then the receiver turns green."

```
self north.
[self center y > 100]
 whileTrue:
 [self go: 10
 Transcript show: '# ' , self center y asString; cr]
```

**Method 19-4.** *Placing transcripts in the method upTo100*

**upTo100**

"Turn the receiver north and move it ten pixels at a time  
until its y-coordinate is less than 100. Then the receiver turns green."

```
self north.
[Transcript show: 'c ' , self center y asString; cr.
 self center y > 100]
 whileTrue:
 [Transcript show: '* ' , self center y asString; cr.
 self go: 10].
self color: Color green
```

Let's compare the traces produced by the different methods. Look in particular at the final values. Observe that `whileTrue:` can be converted to `whileFalse:` by negating the condition (to stay inside while it is raining is logically the same thing as to stay inside while it is false that it is not raining). Use whichever method helps you understand your program better. Try to redefine the method `upTo100` using `whileFalse:` instead of `whileTrue:`. Then define a method that makes the robot move northward one pixel at a time. Compare the exact positions where the robot stops.

Defining conditional loops correctly can be tricky. Keep in mind the following key points (for a `whileTrue: loop`; analogous statements hold for a `whileFalse: loop`):

- The condition is executed *before* the `whileTrue: message` is executed.
- If the condition is true and therefore the conditional block is executed, the condition is then executed again. *Something must happen inside the loop* (such as moving the robot northward) that will eventually make the condition false. Otherwise, the loop will repeat and repeat and repeat forever.
- If the condition is false the first time that it is executed, the loop will never be executed.

It is easy to forget to check the condition carefully with regard to the second bullet item above, namely, that the condition will eventually become false (or true in the case of a `whileFalse: loop`). If it does not, then the loop will repeat endlessly. In writing a conditional loop, you should always keep in mind that the loop should somehow be working toward its termination. That is, the loop should be tending toward a situation that makes the condition false for a `whileTrue: loop` or true for a `whileFalse: loop`.

To investigate the point in the third bullet item above, try the following experiment: Move the robot by using the black halo close to the top edge of the window so that its *y*-coordinate is less than 100. Then invoke the method `upTo100`. As you see, nothing happens, which is as expected: the method is invoked, and the condition `self.center.y > 100` is false, because the position of the robot is smaller than 100. Therefore, the conditional block is not executed.

## Stopping an Infinite Loop

It is not difficult to write an endless loop. You will find yourself in one if the condition expression never returns true for `whileFalse:` or never returns false for `whileTrue:`.

If find that you have written and executed an endless loop, you can stop it by pressing Command+period on a Mac and either Alt+period or Control+C on other platforms. Once you have stopped the loop, you have to figure out why it didn't terminate by opening the debugger by clicking the **Debug** button in the window that appears. You can also print and analyze information using Transcript.

---

**Important!** If a conditional loop is executed once, it will loop endlessly if the conditional block does not perform some action that will eventually break the condition, that is, make a `whileFalse: condition` true or a `whileTrue: condition` false.

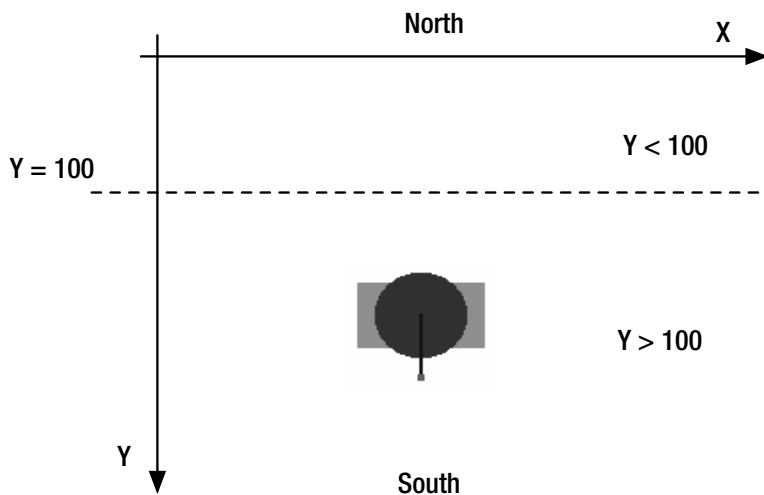
---

Let's consider this difficult point using our robot example. If the loop is to terminate, then the distance between the robot and the horizontal line having *y*-coordinate 100 must become smaller and smaller until the robot crosses the line where the *y* value is 100, and so for the loop to be able to terminate, the expressions inside the conditional block should somehow reduce this distance. More precisely, as long as the robot's *y*-coordinate is too big ( $>= 100$ ), the loop will continue. So to be sure that the loop will terminate, the conditional block must somehow reduce the robot's *y*-coordinate. The expression `self.go: 10` in the conditional



block does this, because the robot's  $y$ -coordinate gets smaller as the robot moves north (and the method makes sure right at the beginning that the robot is in fact pointing north).

Let's look at the case shown in Method 19-5, `upTo100Infinite`. Here the condition is, as in the previous methods, to stop the loop when the  $y$ -coordinate of the robot is less than 100, that is, when the robot arrives above the horizontal line 100 pixels from the top of the screen. However, since this method points the robot so that it is heading south, as shown in Figure 19-5, there is no way for its  $y$ -coordinate to become less than 100 if it begins south of the line with  $y$ -coordinate 100. In such a case, Method 19-5 will never terminate, because the conditional block cannot change the condition to be false. The problem here is that the conditional block *increases* the value of the  $y$ -coordinate, and so the possibility for the condition to evaluate to false actually becomes less with each repetition. This example is rather extreme, but it illustrates clearly the problem of failing to specify a loop that terminates.



**Figure 19-5:** The situation for the execution of the method `upTo100Infinite`

**Method 19-5.** A situation that can lead to an infinite loop

`upTo100Infinite`

```
"Turn the receiver south and move it ten pixels at a time
until its y -coordinate is less than 100. Then the receiver turns green.
```

```
self south.
[self center y > 100]
 whileTrue: [self go: 10].
self color: Color green
```

---

**Important!** When you are defining a loop, always ask yourself whether there is a possibility that the condition might never be broken. If the condition does not have a chance to fail, the loop will continue to execute forever.

---

## Deeper into Conditional Loops

The astute reader may have noticed that the condition in a conditional loop is a block, since it is an expression surrounded by square brackets. A consequence of this is that the condition of a conditional loop is not limited to containing just a single expression. It can contain a sequence of expressions in a block as long as the last message of the condition returns either true or false. This allows you to express more complicated conditional loops. The following “script” shows such a block in outline form. Note that the receiver of the conditional message is a block, in which the receiver does something, then some object does something, and finally a condition is evaluated: is the receiver still working?

```
[self doThis.
 anObject doThat.
 self isStillWorking]
 whileTrue:
 [self grumbleAndKeepOnWorking]
```

With this option, you could change the method `upTo100` to look like Method 19-6. While this method looks nearly the same as `upTo100`, it can have a different effect under certain circumstances. Try to understand what the difference is. For example, add a trace or invoke the debugger in Method 19-6 and analyze it.

**Method 19-6.** *A modified method for moving the robot north*

**notTheSameUpTo100**

"Turn the receiver north and move it ten pixels at a time until its *y*-coordinate is less than 100. Then the receiver turns green.

```
self north.
[self go: 10.
 self center y > 100]
 whileTrue: [].
self color: Color green
```

Since the condition is *always executed at least once*, the robot will always move at least ten pixels, even if its *y*-coordinate is smaller than 100 at the outset. This was not the case in our earlier versions of this method.

## A Simple Interactive Application

Imagine that you want to let the user decide how many steps of length ten pixels east and then ten pixels north a robot should take. Script 19-2 shows how this can be done. The user is prompted for the number of steps; then a branching conditional expression (like those in the previous chapter) is used, in which the condition checks whether the user’s input was a valid number (method `isAllDigits`). If the input is valid, it is converted into a number (`asNumber`), and the robot takes that number of steps. If the input is not valid, the script ends and the robot does not take any steps.

**Script 19-2.** *An interactive staircase with user input*

```

| pica |
answer := (FillInTheBlank
 request: 'Number of steps'
 initialAnswer: '15').
answer isAllDigits
 ifTrue: [pica := Bot new.
 answer asNumber timesRepeat:
 [pica
 go: 10 ;
 north ;
 go: 10 ;
 east]]

```

But we can do better! Script 19-3 shows a wonderful use of conditional loops to ask the user to input a value and then keep asking (loop) until a valid input is given. Once again, we will prompt the user to input a string that represents the number of steps. Once again, we will check using `isAllDigits` whether the data that the user entered represents a number. But this time, the request for user input is contained in the condition block, and if the input does not represent a valid number (`...answer isAllDigits] whileFalse:`), the user is again prompted for input. This loop will run until a string that represents a number is input.

**Script 19-3.** *An interactive staircase with a user input loop*

```

| pica answer |
[answer := (FillInTheBlank
 request: 'Number of steps'
 initialAnswer: '10').
 answer isAllDigits] whileFalse: [].
pica := Bot new.
answer asNumber timesRepeat:
 [pica
 go: 10 ;
 north ;
 go: 10 ;
 east]

```

## When to Use Square Brackets

You have no doubt noticed that I have presented conditions that require the use of empty square brackets [ ] at different places. You may also have noticed that I placed square brackets around the condition (or condition block) in the `whileTrue:` and `whileFalse:` conditional expressions that have appeared in this chapter. So, when do you need to use square brackets? There are basically two rules in Smalltalk: you surround an expression or a sequence of expressions with [ and ] in the following cases:

- You need to execute the same expression a given number of times. For example:
  - `timesRepeat: [ pica go: 10; turnLeft:90 ]` repeats the message sends `pica go: 10; turnLeft:90` four times.

Note that the number of times the expression is repeated can be 1 or even zero, but you will still need the brackets: `1 timesRepeat: [ self go: 120 ]`.
- An expression is executed a variable number of times. For example,
  - `distance < 200 ifTrue: [ self color: Color red ]` executes `self color: Color red` only under certain circumstances,
  - `[ self center y > 100 ] whileTrue: [ self go: 10 ]` repeats conditionally both `self center y > 100` and `self go: 10` multiple times. Therefore, the receiver and the argument are blocks.

To be precise, here is the real definition of when brackets are necessary: the argument of a conditional message (`ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, `timesRepeat:`) or a conditional loop message (`whileTrue:`, `whileFalse:`) is enclosed in square brackets. The receiver of a conditional loop message (`whileTrue:`, `whileFalse:`) is enclosed in square brackets.

## Summary

- A conditional loop consists of a condition and a conditional message, whose argument is a conditional block containing a sequence of expressions.
- The methods `whileTrue:` and `whileFalse:` allow you to define conditional loops in which the conditional block is repeated while the condition holds (`whileTrue:`) or does not hold (`whileFalse:`).
- When you are defining a loop, always ask yourself whether there is a possibility that the condition might never be met, in which case the loop will never be executed. On the other hand, if the condition does not have a chance to fail, the loop will repeat indefinitely.
- Surround an expression or sequence of expressions with square brackets when (1) you need to execute the same expression several times (`4 timesRepeat: [ self go: 10 ]`) or (2) the expression is conditionally executed (`dist < 200 ifTrue: [ self color: Color blue ]`). Also, the receiver of a conditional loop message (`whileTrue:`, `whileFalse:`) is enclosed in square brackets.

Here is a description of the methods introduced in this chapter:

| Method                                                                    | Description                                                                                                                                                                                                 | Example                                                                                                                     |
|---------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| [ <i>aCondition</i> ] <b>whileFalse:</b><br>[ <i>SequenceOfMessages</i> ] | Execute <i>aCondition</i> . If it is false, execute <i>SequenceOfMessages</i> and repeat this step. If <i>aCondition</i> is true, pass to the next expression without executing <i>SequenceOfMessages</i> . | [ answer := (FillInTheBlank<br>request: 'Number of steps'<br>initialAnswer: '10').<br>answer isAllDigits ] whileFalse: [ ]. |
| [ <i>aCondition</i> ] <b>whileTrue:</b><br>[ <i>SequenceOfMessages</i> ]  | Execute <i>aCondition</i> . If it is true, execute <i>SequenceOfMessages</i> and repeat this step. If <i>aCondition</i> is false, pass to the next expression without executing <i>SequenceOfMessages</i> . | [ self center y > 100 ]<br>whileTrue: [ self go: 10 ]                                                                       |



# Boolean and Boolean Expressions

**A**s I mentioned in the two previous chapters, conditional expressions and conditional loops require expressions that return a value that is either *true* or *false*. Such expressions are called *Boolean* expressions, and now we are going to look at them in greater detail, because Boolean expressions and Boolean values are key concepts in programming and indeed in all of computer science. I will show you how to write basic Boolean expressions and how you can combine them to express complex conditions. Finally, I will present some of the most common errors that can arise from missing parentheses.

## Boolean Values and Boolean Expressions

A *Boolean expression* is an expression that returns one of the two values *true* and *false*. Such values are called Boolean.<sup>1</sup> A Boolean value can be *only* true or false. In programming languages, Boolean values are important because they serve as the basis for conditional execution.

### Boolean Values

Boolean values represent the truth or falsity of statements. Here are some true statements: *2 + 2 equals 4*; *the earth accomplishes a complete rotation around its axis in approximately 24 hours*. Here are some false statements: *The French Revolution ended in 1648*; *56 < 34*. In Smalltalk, there are two objects available to represent Booleans: true and false. The object true represents the statement “it is true,” and the object false represents “it is false.” The objects true and false understand all the key messages that allow you to use Boolean expressions, as you will see in a minute.

### Boolean Expressions

A Boolean expression is an expression that returns a Boolean object (true or false). The expression `(2 > 1)` is a Boolean expression. It returns a Boolean object. You can think of a Boolean expression as a question whose answer is true or false (is 2 greater than 1? true!). Table 20-1 shows some examples of Boolean expressions and the kinds of questions they express. Try to print the expression `Time now > (Time new hours: 8)`. You will get either true or false depending on the time you execute it.

**Table 20-1.** Examples of Simple Boolean Expressions

| Expression                                                                                                                  | Explanation                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Bot new color = Color red</code>                                                                                      | Is the color of the newly created robot red?                                                                                                     |
| <code>Bot new center = (100@153)</code>                                                                                     | Is the newly created robot located at the position 100 pixels to the right of the left edge of the screen and 153 pixels down from the top edge? |
| <code>Time now &gt; (Time new hours: 8)</code>                                                                              | Is the time now after 8 o'clock a.m.?                                                                                                            |
| <pre>  pica   pica := Bot new. pica go: 100. (Rectangle origin: 100@200 corner: 300@400)   containsPoint: pica center</pre> | Is the center of the robot inside the rectangle whose top left and bottom right corners are determined by the arguments 100@200 and 300@400?     |

1. The word *Boolean* is in honor of George Boole (1815–1864), a British mathematician and logician who discovered that logical propositions could be represented as symbolic expressions (now called Boolean expressions) and manipulated as mathematical objects.

Observe that the first two questions could be answered from just the information in the Boolean expression (because newly created robots have default colors and locations). The others require knowing what has happened in a script prior to the appearance of the Boolean expression. Evaluate and print the results of the Boolean expressions in the table. After you try each example, experiment by changing the expression and checking your new prediction.

Simple Boolean expressions are based on several messages: `=`, which returns true or false depending on whether two objects are equal; `~=`, which returns true or false depending on whether two objects are not equal; and other messages such as `>`, `<=`, `<`, `>=`, which return true or false depending on whether two objects are in certain order relations.

## Combining Basic Boolean Expressions

The expressions presented in the previous sections are simple, and they are often not sufficient by themselves. However, they can be combined to express complex conditions. *Compound* Boolean expressions can be put together from simpler ones using three logical connectives: *negation* (not), *conjunction* (and), and *alternation* (or). Note that negation does not really combine Boolean expressions, but it is common to present it with conjunction and alternation.

In Smalltalk there are three messages, corresponding to the three logical connectives, that build compound Boolean expressions from simpler ones: `not` for negation (not), `&` for conjunction (and), and `|` for alternation (or). To compose compound expressions, you just send any of these three messages to a Boolean expression, with another Boolean expression as argument in the case of conjunction and alternation. The messages are used like this:

```
aBooleanExpression not
aBooleanExpression & anotherBooleanExpression
aBooleanExpression | anotherBooleanExpression
```

Table 20-2 presents some examples of compound Boolean expressions. Now let us look in detail at how to form compound Boolean expressions by combining simple Boolean expressions using the messages `not`, `|`, and `&`.

**Table 20-2.** Examples of Compound Boolean Expressions

| Expression                                                                                                                                    | Explanation                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>(Bot new color = Color red) not</code>                                                                                                  | Is the color of a newly created robot other than red?                                                                                             |
| <code>  pica  <br/>pica := Bot new.<br/>(pica center = (100@100)) &amp;<br/>(pica direction = 90)</code>                                      | Is a newly created robot located at the position 100@100 and also pointing north?                                                                 |
| <code>Time now &gt; (Time new hours: 8)  <br/>(Date today weekday asString = 'Sunday')</code>                                                 | Is the time now after 8 a.m. or is it Sunday (or both)?                                                                                           |
| <code>Time now &gt; (Time new hours: 8)  <br/>(Date today weekday asString = 'Sunday') not</code>                                             | Is the time now after 8 a.m. or is it not Sunday (or both)?                                                                                       |
| <code>((pica color = Color red) &amp;<br/>(pica direction = 90))  <br/>((pica color = Color blue) &amp;<br/>(pica direction = 90) not)</code> | Is it true that <i>either</i> pica's color is red and he is pointing north <i>or</i> pica's color is blue and he is not pointing north (or both)? |



## Negation (not)

Negation is used to express the contrary of something. In Smalltalk, negation is expressed using the message `not`, which simply negates the Boolean expression to which it is sent. In the last line of Script 20-1, the message `not` is sent to the expression `(aBot color = Color red)`. If such an expression is true, then its negation will be false, and vice versa. You can interpret `(aBot color = Color red) not` as follows: “Is the color of the robot something other than red?” or “Is it false that the color of the robot is red?”

### Script 20-1. Example of negation

```
| aBot |
aBot := Bot new.
aBot color: Color green.
(aBot color = Color red) not
“Is the color of the robot something other than red?”
```

Note that you can always negate (logically reverse) a condition to switch from one form to the other. For example, in Method 20-1, `distanceFromCenter >= 200` is the negation of the expression `distanceFromCenter < 200`, as shown in Method 20-2. So these two methods have exactly the same effect. Again, I suggest that you add a trace to understand what is happening.

**Method 20-1.** *If a robot's distance to the center is not greater than or equal to 200 (that is, less than 200), change its color to red.*

```
redWhenCloseToCenter

| distance |
distance := self distanceFrom: World center.
distance >= 200
iffalse: [self color: Color red]
```

**Method 20-2.** *If a robot's distance to the center is less than 200, change its color to red.*

```
redWhenCloseToCenter

| distance |
distance := self distanceFrom: World center.
distance < 200
iftrue: [self color: Color red]
```

## Conjunction (and)

The term *conjunction* literally means *joined together*. Conjunction is used to express that a compound Boolean expression `expression1 & expression2` is true only when *both* of the two Boolean subexpressions `expression1` and `expression2` are true. In Squeak, a conjunction is defined by sending the binary message `&` to a Boolean expression with another Boolean expression as argument. Again, a conjunction is true only when both subexpressions that compose it

are true. If either or both subexpressions are false, then the compound subexpression is false. In Table 20-3, the compound expression will be true only if *both* (aBot center = 100@100) *and* (aBot direction = 90) are true.

**Table 20-3.** *An Example of Conjunction*

| Expression                                      | Explanation                                                  |
|-------------------------------------------------|--------------------------------------------------------------|
| (aBot center = 100@100) & (aBot direction = 90) | Is the robot located at position 100@100 and pointing north? |

## Alternation (or)

Alternation is used to express the idea of choice. Think of an alternative: do you want coffee or tea? cake or ice cream? An alternation is defined by sending the binary message | to a Boolean expression with another Boolean expression as argument. An alternation is used to express that you are asking whether *at least one* of the Boolean expressions is true. Therefore, an alternation is true if one or both of the expressions it is composed of is true.

This definition of alternation often causes confusion. That is because in English, the word “or” is used in *two different ways*. When your host at a dinner party, Fred, asks, “Would you like coffee or tea,” he is probably expecting one of the following answers: “coffee,” “tea,” “nothing, thank you.” But if he asks, “Would you like some ice cream or cake,” he is probably expecting you to say one of “ice cream,” “cake,” “both, please!” When “or” means either the one or the other *but not both*, we call it *exclusive or*. When “or” means one or the other *or both*, we call it *inclusive or*. In Smalltalk, as in most computer programming languages, “or” means *inclusive or*.

In Table 20-4, the compound expression is true if the expression (aBot center = 100@100) is true *or* the expression (aBot direction = 90) is true *or* both are true.

**Table 20-4.** *An Example of Alternation*

| Expression                                        | Explanation                                                                |
|---------------------------------------------------|----------------------------------------------------------------------------|
| (aBot center = (100@100))   (aBot direction = 90) | Is the robot located at position 100@100 or is it heading north (or both)? |

## All of the Above

The last two examples of Table 20-2 show that you can combine Boolean expressions multiple times, negate them, and group them using alternation (or) and conjunction (and) to represent complex conditions.

## Some Smalltalk Points

Recall that classes are factories for producing objects. When a particular robot is created by the class Bot, we say that the robot is an *instance* of the class Bot. In Smalltalk, Boolean values are also objects. The object true is an instance of the class True, which defines the behavior of the Boolean value true. Similarly, false is an instance of the class False, which defines the behavior of the Boolean value false. Note that even if true and false are objects in the same

sense that a robot created by the class `Bot` is an object, they are so central to Smalltalk that `true` and `false` are special objects. Hence, you do not have to create them using `new`. Instead, `true` and `false` exist all the time, and you do not have to worry about their creation. Note that the Boolean objects `true` and `false` start with a lowercase letter.

As I explained in the first section of this chapter, when you evaluate an expression such as `Time now > (Time new hours: 8)` you get either the Boolean object `true` or the Boolean object `false`, depending on the time you execute it. I also said that to compose Boolean expressions, the messages `&`, `not`, and `|` are sent to Boolean expressions, and that the result of a Boolean expression is either `true` or `false`. Therefore, the messages `&`, `not`, and `|` are methods defined on the classes `True` and `False`, which manufacture the objects `true` and `false`.

Table 20-5 shows how the three Boolean operations are used.

**Table 20-5.** *The Three Boolean Operations*

| Kind              | Message | Examples                                                                                  | Result        |
|-------------------|---------|-------------------------------------------------------------------------------------------|---------------|
| Negation (not)    | not     | <code>falseExpression not</code>                                                          | true          |
|                   |         | <code>falseExpression not</code>                                                          | false         |
|                   |         | <code>(Bot new color = Color red) not</code>                                              | true          |
| Conjunction (and) | &       | <code>trueExpression &amp; trueExpression</code>                                          | true          |
|                   |         | <code>falseExpression &amp; trueExpression</code>                                         | false         |
|                   |         | <code>trueExpression &amp; falseExpression</code>                                         | false         |
|                   |         | <code>falseExpression &amp; falseExpression</code>                                        | false         |
|                   |         | <code>(aBot center = 100@100) &amp; (aBot direction = 90)</code>                          | true or false |
| Alternation (or)  |         | <code>trueExpression   trueExpression</code>                                              | true          |
|                   |         | <code>falseExpression   trueExpression</code>                                             | true          |
|                   |         | <code>trueExpression   falseExpression</code>                                             | true          |
|                   |         | <code>falseExpression   falseExpression</code>                                            | false         |
|                   |         | <code>Time now &gt; (Time new hours: 8)   (Date today weekday asString = 'Sunday')</code> | true or false |

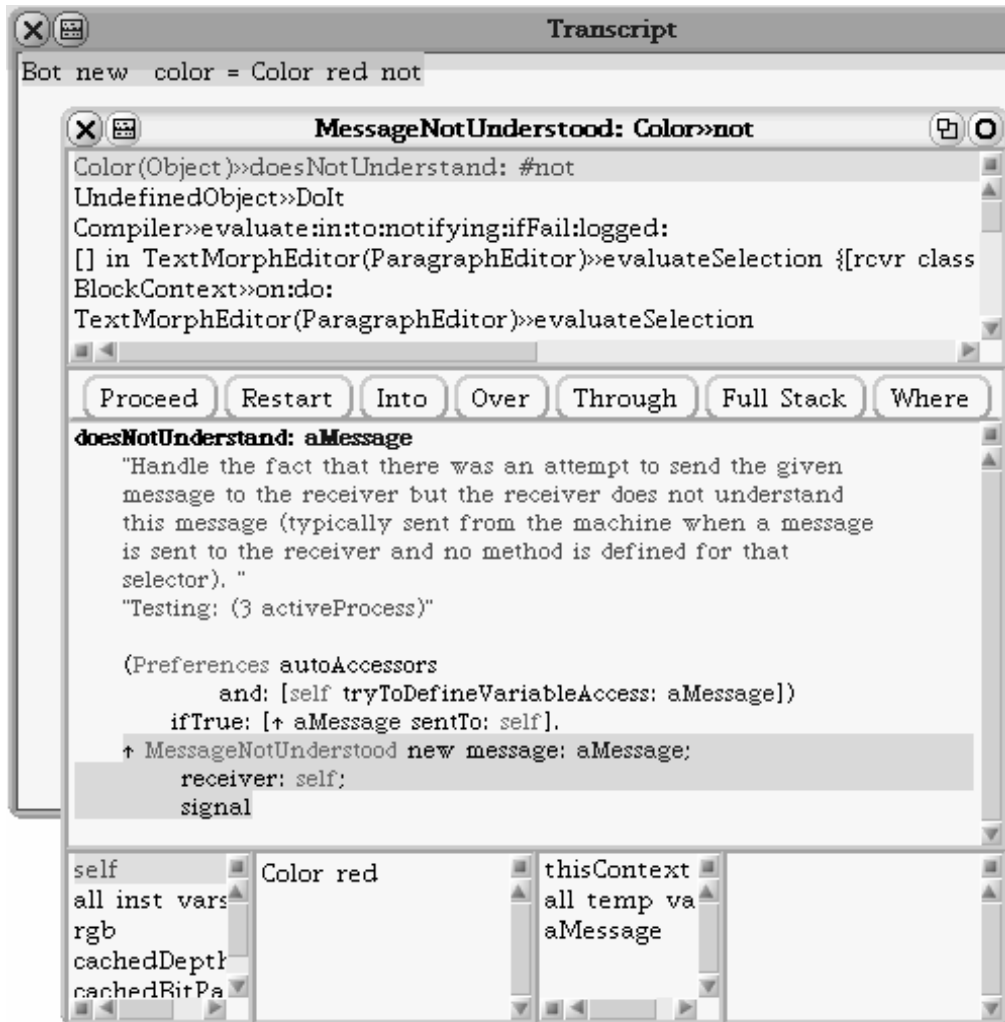
## Missing Parentheses (a Frequent Mistake)

Every now and then you might find yourself in trouble with the syntax of Smalltalk. All programmers, even experienced ones, have such trouble on occasion. The difference between a beginner and an experienced programmer is not that one makes mistakes and the other doesn't. The main difference is that an experienced programmer is much more adept at identifying errors and fixing them.

Missing parentheses is a frequent source of mistakes in both logic and syntax, and so I am going to show you how to analyze the errors you may make. Basically, when you are composing a Boolean expression, you have to identify clearly the expression to which the messages `not`, `|`, and `&` are sent. Let us look at a particular case.

## A Case Study

Script 20-2 shows a Boolean expression that fails to represent the following question: “Is the color of a newly created robot different from red?” To see that executing this script leads to an error, execute the expression described in Script 20-2, open the debugger when the error occurs, and select the first line in the top pane to obtain Figure 20-1.



**Figure 20-1.** The message not is not sent to the complete Boolean expression `Bot new color = Color red`, but is sent instead to the expression `Color red`, which returns a color. But color objects do not understand the message not, and therefore an error occurs.

**Script 20-2.** Missing parentheses cause misidentification of the receiver of a not message.

```
Bot new color = Color red not
```

## Using the Debugger

The title window of the debugger already gives us some information: `MessageNotUnderstood: Color»not`. This tells us that a color object (an object created by the class `Color`) does not understand the message `not`.

Now when you select the topmost line of the top pane, you see the body of the method `doesNotUnderstand:` in the second pane. When you click on `self` in the left bottom pane, you see (in the second pane on the bottom) that the receiver is not a `Boolean`, as it should be, but a color, `Color red`. If you click on `aMessage` on the right bottom pane, you will see which message was not understood. In our case, you will see `not`, which means that the message `not` has not been sent to the right receiver, because it was sent to the result of the expression `Color red`, which is a color object and does not understand the message `not`.

## Understanding the Problem

The reason that the message `not` was sent to the expression `Color red` and not to the complete Boolean expression is related to the way Smalltalk executes expressions, as explained in Chapter 11. Recall that first, expressions surrounded by parentheses are executed, then unary messages, then binary, and finally keyword-based messages. In our case, the message `not` is a unary message. Therefore, it is evaluated before the binary message `=`, and so it is sent to the result of the expression `Color red`. To get the correct execution order, you have to surround the appropriate expression in parentheses, as shown in Script 20-3. Now the message `not` will be sent to the result of the `=` message.

The order of execution of the messages in the defective Script 20-2 is as follows: The expression `Bot new color = Color red not` is executed as though it were written fully parenthesized as follows: `((Bot new) color) = ((Color red) not)`. Therefore, first both parts of the binary method `=` are evaluated, that is, the expression `((Bot new) color)`, which returns the new robot's color, and the expression `((Color red) not)`. The execution of the expression `((Color red) not)` first evaluates `Color red`, which returns a color object, and then the message `not` is sent to this color object, which leads to an error.

To obtain the desired behavior, the expression should be parenthesized so that the `not` message is sent to the result of the `=` message. The expression is then as shown in Script 20-3.

**Script 20-3.** *Using parentheses helps to ensure that your message is formulated correctly.*

```
(Bot new color = Color red) not
```

This expression is executed as follows: first both parts of the binary method `=` are evaluated. The first returns a color object that represents the color of the new robot, while the second returns a red color object. These two color objects either are or are not the same. Then the `=` message is executed, which sends the result of the right-hand expression to the color object for the robot's color. The execution of the message `=` returns a `Boolean`, to which the message `not` is sent.

If you are ever unsure about the order in which messages are executed, you should use the fact that expressions in parentheses are executed before other expressions. Therefore, you can put parentheses around expressions to make sure that your messages are executed in the correct logical order.

## Similar Problems and Solutions

It would be tedious to try to examine all the many similar problems you may encounter with your Boolean expressions. Try executing Scripts 20-4 and 20-6 and see whether you can understand what is wrong. The corresponding correctly parenthesized expressions appear in Scripts 20-5 and 20-7.

**Script 20-4.** *Missing parentheses result in misidentification of the receiver of &.*

```
| aBot |
aBot := Bot new.
aBot center = 100@100 & aBot penSize = 5
```

**Script 20-5.** *Parentheses are used to get the proper receiver of &.*

```
| aBot |
aBot := Bot new.
(aBot center = 100@100) & (aBot penSize = 5)
```

**Script 20-6.** *Missing parentheses result in misidentification of the receiver of |.*

```
| aBot |
aBot := Bot new.
aBot center = 100@100 | aBot direction = 90
```

**Script 20-7.** *Parentheses are used to get the proper receiver of |.*

```
| aBot |
aBot := Bot new.
(aBot center = 100@100) | (aBot direction = 90)
```

## Summary

- The Boolean values in Smalltalk are true and false. The value true represents a true statement, and false a false statement.
- Boolean expressions are expressions that return Boolean values.
- Compound Boolean expressions are composed of simple Boolean expressions using conjunction (and), alternation (or), and negation (not).

The table below reviews the three Boolean operations.

| Kind              | Message | Examples                                                                     | Result        |
|-------------------|---------|------------------------------------------------------------------------------|---------------|
| Negation (not)    | not     | <i>falseExpression</i> not                                                   | true          |
|                   |         | <i>falseExpression</i> not                                                   | false         |
|                   |         | (Bot new color = Color red) not                                              | true          |
| Conjunction (and) | &       | <i>trueExpression</i> & <i>trueExpression</i>                                | true          |
|                   |         | <i>falseExpression</i> & <i>trueExpression</i>                               | false         |
|                   |         | <i>trueExpression</i> & <i>falseExpression</i>                               | false         |
|                   |         | <i>falseExpression</i> & <i>falseExpression</i>                              | false         |
|                   |         | (aBot center = 100@100) &<br>(aBot direction = 90)                           | true or false |
| Alternation (or)  |         | <i>trueExpression</i>   <i>trueExpression</i>                                | true          |
|                   |         | <i>falseExpression</i>   <i>trueExpression</i>                               | true          |
|                   |         | <i>trueExpression</i>   <i>falseExpression</i>                               | true          |
|                   |         | <i>falseExpression</i>   <i>falseExpression</i>                              | false         |
|                   |         | Time now > (Time new hours: 8)  <br>(Date today weekday asString = 'Sunday') | true or false |



# Coordinates, Points, and Absolute Moves

**U**p to now, the messages sent to a robot telling it to move gave instructions relative to the robot's current position. For example, the expression `pica go: 100` tells `pica` to go forward 100 pixels from his *current* position in his *current* direction. Such a move is said to be *relative* because the position reached by the robot at the end of the move depends on its initial position. This kind of move is very useful, as we have seen, but sometimes, you would like to be able to tell a robot to move to a specific location on the screen, such as the screen's center. Such a move, in which the robot ends up at a certain position irrespective of where it started, is called *absolute*. For making absolute moves we need a coordinate system, which is a way to represent a specific location on the screen.

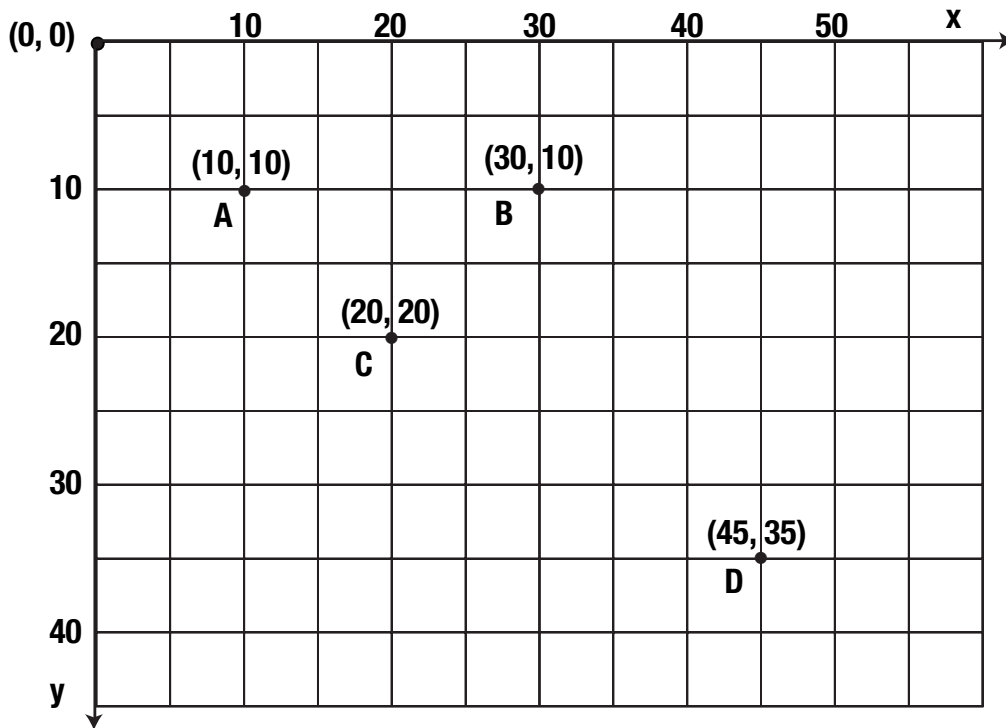
You have encountered this coordinate system a bit in earlier chapters, but now it is time for more detail. You may have encountered coordinate systems in your study of mathematics. Furthermore, you use a coordinate system when you are looking for a street on a map. The map listing may indicate that a given street is located in a square identified by a letter and a number, or perhaps by two letters or even two numbers. The same applies to your computer screen. You can refer to a point on the screen by giving a pair of numbers, one for the horizontal direction and the other for the vertical direction.

In this chapter I will introduce the precise definitions of points and coordinates. Then I will present some new robot behavior and give you some experiments to try. Knowledge of points and coordinates will help you to explore new problems in the future, such as using robots to model animal behavior.



## Points

Since everything in Smalltalk is an object, locations on the screen are *also* described by objects, called *points*. Points are created by the class `Point`, and their behavior is similar to that of the points you are familiar with from mathematics. In two dimensions, a point is composed of two coordinates: the *x*-coordinate, for the horizontal direction, and the *y*-coordinate, for the vertical. A point is created by sending the message `@` to a number with another number as argument. For example, point D of Figure 21-1 is created by the expression `45@35`. Its *x*-coordinate is 45, and its *y*-coordinate is 35.



**Figure 21-1.** A point in Smalltalk represents a location in two dimensions. A point has an *x* (horizontal) coordinate and a *y* (vertical) coordinate.

---

**Important!** `200@400` is a point whose *x*-coordinate is 200 and whose *y*-coordinate is 400.

---

Script 21-1 shows how to access the individual components of a point.

**Script 21-1.** *Accessing the components of a point*

```
| point1 |
point1 := 45@35.
point1 x
-> 45
point1 y
-> 35
```

It is worth noting that the coordinate system in Smalltalk is not quite the same as the mathematical one. Figure 21-1 shows that in contrast to the standard mathematical model, the positive direction for the  $y$ -axis runs from the top of the screen to the bottom. However, the  $x$ -axis increases from left to right, as in standard mathematical notation. We say in Squeak that the *origin* of the coordinate system is at the top left corner of the screen. Thus, the point 45@35 is located 45 pixels to the right of the left edge of the screen and 35 pixels down from the top. Points with one or both coordinates negative are located somewhere off the screen.

---

**Important!** The Smalltalk coordinate system has its origin (0,0) at the top left corner of the screen, and the positive direction of the  $y$ -axis is downward, from the top of the screen to the bottom.

---

All the usual mathematical operations are available for points. In this chapter I will present only a few of operations that you will be using in the future. For example, you can multiply a point by a number to obtain a point whose coordinates are those of the initial point multiplied by the number. Thus  $(100@200) * 3$  yields the point 300@600. You can also use such common mathematical operations as addition and subtraction on points themselves. The operations are carried out coordinatewise, and thus  $(100@200) + (40@360)$  yields the point (140@560). Note that binary operations such as  $*$  and  $+$  can operate on points to create new points. Script 21-2 shows these operations.

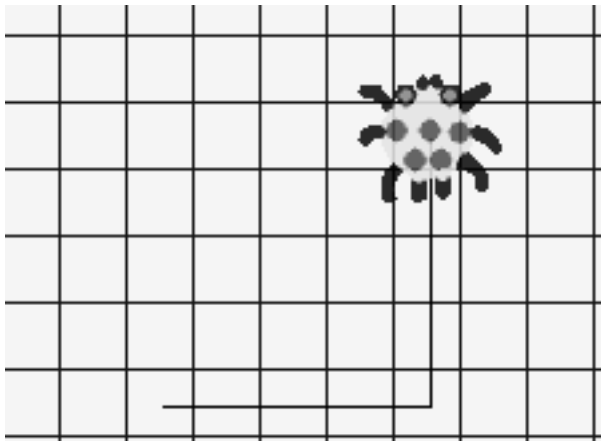
**Script 21-2.** *Manipulating points to create new points*

```
| point1 point2 point3 |
point1 := 200@400.
point2 := point1 * 2
point2
-Printing the returned value: 400@800
point2 x
-Printing the returned value: 400
point2 y
-Printing the returned value: 800
point3 := (50@60) + point1.
point3 x
-Printing the returned value: 250
point3 y
-Printing the returned value: 460
point1 + 85
-Printing the returned value: 285@485
"85 is shorthand for the point 85@85"
```

## Using Grids

To assist you in understanding points, you can have a look at the information displayed in the balloon that pops up when you let the mouse hover over a robot. You can also use a grid. Squeak can actually draw a grid on the background of the screen. To obtain a grid, bring up the **world** menu, select **appearance...**, and then the menu item **use standard texture** or **make graph paper....** Choosing the latter lets you define the size and color of the grid.

As shown in Script 21-3 and illustrated in Figure 21-2, you can also program the grid using the following methods: `drawGrids` and `undrawGrids` to draw and erase grids, `gridColor: aColor` to change the color of a grid, `gridSize: anInteger` to specify the size of a grid (the number of pixels for the side length of each square), `gridWorldColor: aColor` to change the color of the world when a grid is drawn, and `worldColor: aColor` to change the color of the world. You can also retrieve the size of a grid using the method `gridSize`.



**Figure 21-2.** A grid of size 25 pixels

### Script 21-3. Setting up a grid

```
| env |
env := BotEnvironment default.
env gridSize: 25.
env gridWorldColor: Color paleBlue.
env gridColor: Color blue.
env drawGrids
```

Using the **world** menu, you can change the size of the screen. To go into full-screen mode, bring up the world menu, select **appearance...**, and then turn full-screen mode on or off. You can also use the methods `fullScreenOff` and `fullScreenOn` as shown in Script 21-4.

**Script 21-4.** *Setting the screen size*

```
BotEnvironment default fullScreenOn
```

## A Source of Errors with Points

The way points are created and the order in which messages are executed may lead to errors, as shown in the first line of Script 21-5. The message `send 50@60 + 200@400` returns `aB3dVector` instead of a point representing the sum of the two points. The problem is that due to the order of message execution, a *point*, and not an *integer*, is the receiver of the message `@`, and in that case it returns another kind of point (a 3D vector) that does not interest us here. I will explain exactly what went wrong in the next paragraph. But in any case, we certainly didn't get the result we wanted! So once again, let me remind you to pay careful attention to the order in which the messages are sent and use parentheses as necessary.

**Script 21-5.** *A possible error with points*

```
50@60 + 200@400 "returns a 3D vector, not a point"
-> a B3dVector3(250.0 260.0 400.0)
(50@60) + (200@400)
-> 250@460
```

---

**Note** To avoid trouble with points, surround them with parentheses when they are involved in complex operations.

---

Recall from Chapter 11 the order in which messages are executed, and in particular the following:

- Expressions inside parentheses ( ) are evaluated first.
- Unary messages are executed before binary ones, and binary messages are executed before keyword-based ones.
- Messages of the same type are evaluated in order from left to right.

The method `@` is just a binary method like any other, and it has the same priority as binary methods such as `+`, `*`, and `=`. Therefore, the expression `50@60 + 200@400` is executed as though it had been typed as follows: `((50@60) + 200) @ 400`. The second message `@` will end up being sent to a point and not an integer. Let us look at what happens in the first line of Script 21-5, which does not return the point `250@460`, as you might have expected.

## Decomposing `50@60 + 200@400`

As just mentioned, `50@60 + 200@400` is equivalent to `((50@60) + 200) @ 400`.

**Step 1.** `@` is sent to 50 with the argument 60. The point `50@60` is returned.

**Step 2.** `+` is sent to the point `50@60` with the argument 200. The point `250@260` is returned, because when a number is passed to a point as an argument, it is considered as the point having the same value for its *x*- and *y*-coordinates, here `200@200`.

**Step 3.** `@` is sent to `250@260` with 400 as argument, and the object `B3DVector3(250.0 260.0 400.0)` is returned.

If we now simply put parentheses around the two points, we obtain a point that is the sum of those two.

## Decomposing `(50@60) + (200@400)`

**Step 1.** Expressions in parentheses are evaluated first.

**Step 1.1.** `@` is sent to 50 with 60 as argument and the point `50@60` is returned.

**Step 1.2.** `@` is sent to 200 with 400 as argument and the point `200@400` is returned.

**Step 2.** `+` is sent to `50@60` with argument `200@400` and the new point `250@460` is returned.

The moral is this: put parentheses around points when you are manipulating them.

## Absolute Moves

Now that we can specify a location on the screen, we can tell a robot to go directly to that location. For this task we have the two methods `goTo: aPoint` and `jumpTo: aPoint`.

- Sending the message `goTo: aPoint` to a robot tells it to go to the location represented by the point.
- Sending the message `jumpTo: aPoint` to a robot tells it to jump to the location represented by the point.

Note that the messages `jump:` and `jumpTo:` do not leave a trace, while `go:` and `goTo:` do. Let us practice now. Try to figure out what Script 21-6 does. Then try to estimate the size of your screen in pixels by positioning a robot as close as possible to the bottom right corner.

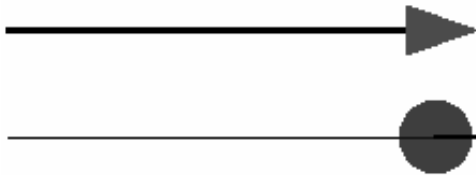
**Script 21-6.** *Going and jumping directly to a location*

```
| pica |
pica := Bot new.
pica goTo: 200@400.
pica jumpTo: 300@400.
pica go: 1.
pica jumpTo: 400@400.
pica goTo: 450@400
```

The following section will stress the differences between the methods `go: aDistance` and `goTo: aPoint`, and `jump: aDistance` and `jumpTo: aPoint`.

## Relative versus Absolute Motion

Now we will look closely at the difference between the methods `go: aDistance` and `goTo: aPoint`. The method `go:` tells a robot to move a given distance *along its current direction*. Thus, where the robot ends up depends on its current location and its current direction. Script 21-7 illustrates this.

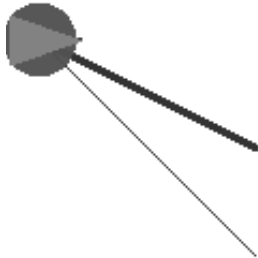
**Script 21-7.** *Parallel motion*

```
| pica marge |
pica := Bot new.
marge := Bot new.
marge lookLikeTriangle.
pica lookLikeCircle.
marge color: Color red.
marge penSize: 3.
marge north.
marge jump: 50.
marge east.
pica go: 200.
marge go: 200.
```

As you can see, the two robots are moving along parallel lines and do not end up in the same location, even though the same message, `go: 200`, was issued to them at the end of the script.

In contrast, the method `goTo: aPoint` tells a robot to move to a fixed location *regardless* of its position and direction before the move. This is illustrated in Script 21-8.

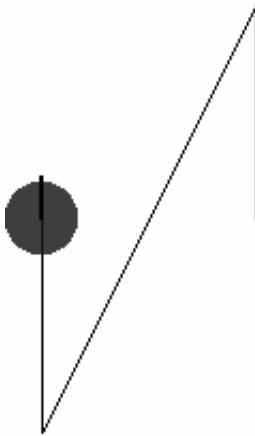
**Script 21-8.** *Convergent motion*



```
| pica marge |
pica := Bot new.
marge := Bot new.
marge lookLikeTriangle.
pica lookLikeCircle.
marge color: Color red.
marge penSize: 3.
marge north.
marge jump: 50.
marge east.
pica goTo: World center - 100.
marge goTo: World center - 100.
```

In this case, the two robots end up at the same location. One says that the method `go:` produces *relative* motion, whereas the method `goTo:` produces *absolute* motion. In the script we used the expression `World center - 100` so that you get exactly the same picture as the one shown here even if your computer monitor has a different resolution from the one used to write this book.

Finally, note that the methods `go:` and `goTo:` do not change the direction of the robot. This is illustrated in Script 21-9. In this script we tell `pica` to move forward 100 pixels from his current position, `go` directly to a position that is located at distance (100, 100) from the center of the screen, and then move forward 100 pixels in his current direction.

**Script 21-9.** *Combining absolute and relative motion*

```
| pica |
pica := Bot new.
pica lookLikeCircle.
pica north.
pica go: 100.
pica goTo: (World center - (100@100)).
pica go: 100.
```

## Some Experiments

Here are some experiments for you to try so that you can become more familiar with the concepts that have been presented here. As you can see, the robot does not change its direction when it is transported to a given location using `goTo:`.

### Experiment 21-1 (Rectangle 1)

Using the methods `goTo:` and `jumpTo:`, define a method `rectangleTopLeft: point1 bottomRight: point2` that draws a rectangle. A message send might look like this: `pica rectangleTopLeft: 200@500 bottomRight: 350@700`.

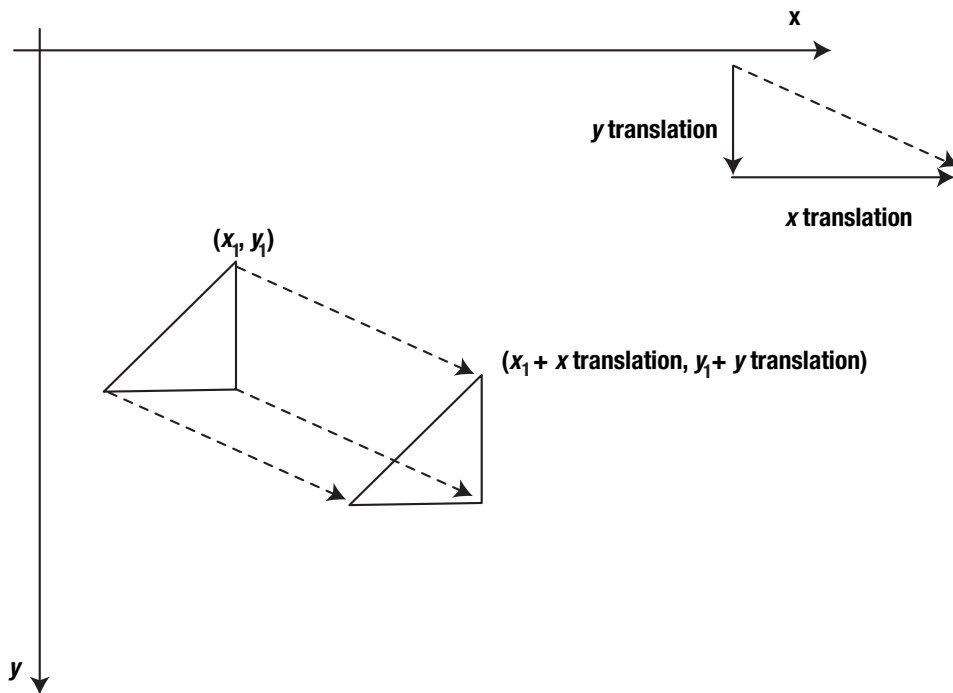
### Experiment 21-2 (Rectangle 2)

Define another method, `rectangleOrigin: point1 extent: point2`, in which the second point no longer represents the opposite corner but the size (base and height) of the rectangle. For example, `pica rectangleOrigin: 200@600 extent: 350@500` draws a rectangle with upper left corner `200@600` and having base of length 350 pixels and height 500.



## Translations

If we move every point of a geometrical figure by the same distance in the same direction, we obtain the same figure, but in another position. This operation is called a *translation* in mathematics. As shown in Figure 21-3, a translation can be thought of as motion by a certain amount in the  $x$ -direction and a certain amount in the  $y$ -direction. And of course, these numbers don't have to be the same. We can therefore represent the translation of a figure as the sum of a vertex and a “translation point” representing the size of the translation in the  $x$ - and  $y$ -directions. Then the vertex of the new figure is equal to the sum of a vertex of the original figure and the translation point. In Figure 21-3, the vertex  $(x_1, y_1)$  is translated by adding a translation point, and the result is a point whose new  $x$ -coordinate is  $x_{new} = x_1 + x \text{ translation}$  and whose new  $y$ -coordinate is  $y_{new} = y_1 + y \text{ translation}$ . For example, the point 200@300 translated by the translation point 50@75 is 250@375.



**Figure 21-3.** Translating a figure requires adding the same  $x$  value and  $y$  value to each point of the figure.

### Experiment 21-3 (Triangle 1)

Define a method named `triangleAt: firstPoint point2: secondPoint point3: thirdPoint` that draws a triangle with vertices at the three points given as arguments. Script 21-10 illustrates how to use such a method.

**Script 21-10.** *Using triangleAt:secondPoint:thirdPoint:*

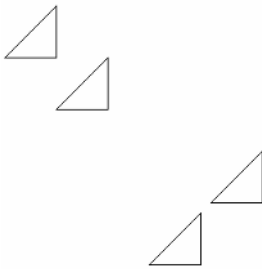


```
| pica |
pica := Bot new.
pica
 triangleAt: 200@300
 point2: 200@250
 point3: 150@300
```

## Translating Triangles

Now that you have a method for drawing triangles, you can draw several identical triangles by simply translating the first one. In Script 21-11, I define three translations and then draw the corresponding triangles.

**Script 21-11.** *Using triangleAt:point2:point3: to draw translated triangles*



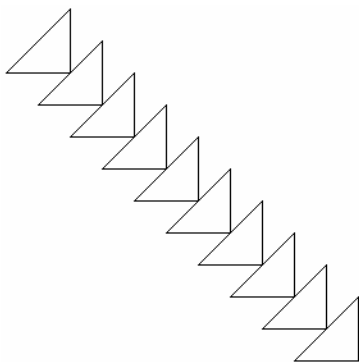
```
| pica firstPoint secondPoint thirdPoint t1 t2 t3 |
firstPoint:= 200@300. "three points of a triangle"
secondPoint:= 200@250.
thirdPoint:= 150@300.
t1 := 50@50. "three points for translating the triangle"
t2 := 90@150.
t3 := 150@90.
pica := Bot new.
pica beInvisible.
pica triangleAt: firstPoint point2: secondPoint point3: thirdPoint.
pica triangleAt: firstPoint + t1 point2: secondPoint + t1 point3: thirdPoint + t1.
pica triangleAt: firstPoint + t2 point2: secondPoint + t2 point3: thirdPoint + t2.
pica triangleAt: firstPoint + t3 point2: secondPoint + t3 point3: thirdPoint + t3.
```

We could also have defined another method `triangleAt:point2:point3:translation:` that performs the translation so we wouldn't have to do the point addition ourselves. Such a solution is safer because it would prevent us from accidentally applying different translations to different points of the same triangle, and therefore I suggest that you implement it.

## Flying Geese

One can repeat the translation operation to obtain repeating patterns. Script 21-12 generates a pattern that looks like *flying geese*. Note that I chose the amount of translation so that each triangle just touches the next and so that they run along a diagonal.

**Script 21-12.** *Flying geese*



```
| pica translation firstPoint secondPoint thirdPoint |
firstPoint:= 200@300.
secondPoint:= 200@250.
thirdPoint:= 150@300.
translation := 25@25.
pica := Bot new.
10 timesRepeat:
 [pica triangleAt: firstPoint point2: secondPoint point3: thirdPoint.
 firstPoint:= firstPoint + translation.
 secondPoint:= secondPoint+ translation.
 thirdPoint:= thirdPoint + translation].
```

Script 21-13 shows how you can write the translation in a more concise way using the fact that points can be multiplied as well as added.

**Script 21-13.** *Flying geese*

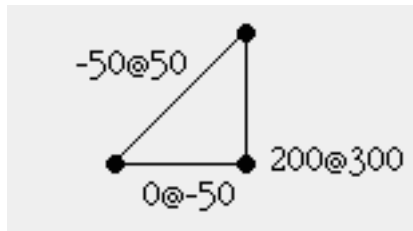
```

| pica times |
pica := Bot new.
times := 1.
10 timesRepeat:
 [pica triangleAt: 200@300
 point2: 200@250
 point3: 150@300
 translation: (25@25) * times.
 times := times + 1].

```

**Experiment 21-4 (Triangle 2)**

As a variation of Experiment 21-3, define a method named `triangleAt: aPoint delta1: aPoint1 delta2: aPoint2` that draws a triangle starting at point `aPoint` and then uses each of the two following arguments as the difference between another point of the triangle and the first point. Thus `triangleAt: 200@300 delta1: 0@50 delta2: -50@50` draws the same triangle as `triangleAt: 200@300 point2: 200@250 point3: 150@300`.



## Absolute Moves at Work

You may be wondering why I am making such a big deal about points. So far, none of the drawings that we have made required points. In fact, executing most of those drawings using points would have been quite difficult. Imagine trying to draw a pentagon using only the `goTo:` message. Nevertheless, absolute positions are useful. The following example illustrates this point about points. We will use a point to keep track of a robot's position at different points during the execution of a complex drawing. Then we will use a saved position to continue our drawing.

The first example is based on Script 3-4 of Chapter 3, in which the letter A was drawn. We noted in that chapter that the bottom half of the right-hand vertical bar of the A is drawn twice by the script. We considered that it was not that big a problem. But imagine a situation in which drawing a line was an expensive proposition, either in terms of computation time or in terms of ink on a printer. In that case, it would be worth the effort to avoid drawing a line twice. My solution is to use a point to store the location of the robot at the left edge of the horizontal bar and then to return to that point to draw the bar after the rest of the letter is drawn.

Script 21-14 draws a letter A as in Script 3-4. I then modify the script to obtain Script 21-15, which draws the A without drawing over a line using an absolute jump.

**Script 21-14.** *Drawing the letter A by going over a line twice*

```
| pica |
pica := Bot new.
pica turnLeft: 90.
pica go: 100.
pica turnRight 90.
pica go: 100.
pica turnRight 90.
pica go: 100.
pica turnRight 180.
pica go: 40.
pica turnLeft 90.
pica go: 100
```

**Script 21-15.** *Drawing the letter A using a jump to an absolute point*

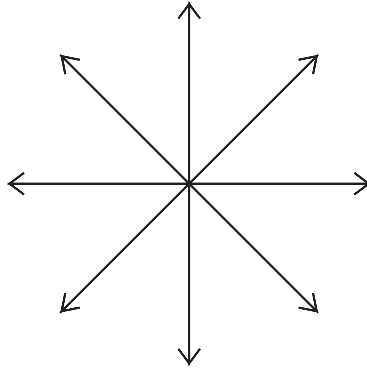
```
| pica barPoint |
pica := Bot new.
pica turnLeft: 90.
pica go: 40.
barPoint := pica center.
pica go: 60.
pica turnRight: 90.
pica go: 100.
pica turnRight: 90.
pica go: 100.
pica jumpTo: barPoint.
pica turnLeft: 90.
pica go: 100
```

In script 21-15, the left-hand vertical bar of the A is drawn in two steps, first 40 then 60 pixels. Between the steps, the absolute location of the robot is obtained using the method `center`, and this location is stored in the variable `barPoint`. This is the location where the bar of the A should be drawn. After the last vertical bar is drawn, the robot jumps back to the location of the bar using the method `goTo: barPoint` and draws the horizontal bar.

By the way, you can verify that the letter A is drawn correctly at any angle by adding a command `turnLeft:` or `turnRight:` through any number of degrees after the creation of the robot.

### Experiment 21-5 (Arrows)

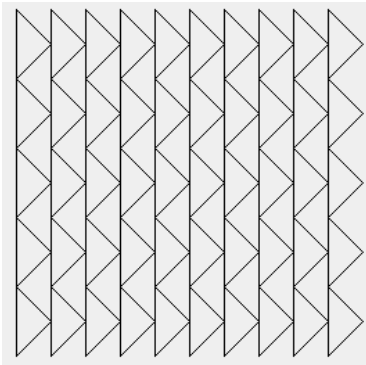
Using the same technique as in Script 21-15, define a script that generates eight arrows shooting from the robot's origin in eight different directions, as shown in the figure below. Hint: First define a method called `arrow: aPoint` that draws an arrow pointing in the current direction starting at the given point. Then use an additional variable to remember the origin of the arrow. Once you have successfully written and tested such a method, I strongly suggest that you solve the same problem using the methods `go:` and `jump:`. This way, you will really understand the difference between two ways of expressing the same problem.



---

## Loops and Translations

Before reading the following, try to define the script that draws Figure 21-4.



**Figure 21-4.** *A pattern of flying geese*

My solution appears in Script 21-16, which draws Figure 21-4. Points define a number of useful methods, among them `negated` and `setX:setY::`

- The method `negated` sent to a point returns a point whose  $x$ - and  $y$ -coordinates are the negated values of the receiving point. Thus, the point `(200@400)` `negated` is the point `-200@-400`. Note that the parentheses are necessary. Indeed, `negated` is also a method understood by numbers. Thus, the expression `200@400 negated` yields the point `200@-400` (a point off the screen) because the method `negated`, being a unary method, is executed by the number `400` before the method `@` is executed. In Script 21-16 this method is used to produce a translation in the opposite direction.
- The method `setX:setY:` changes the coordinates of a point. Thus, after the expression `aPoint setX: 200 setY: 400` is executed, the point `aPoint` has `200` for its  $x$ -coordinate and `400` for its  $y$ -coordinate.

**Script 21-16.** *A pattern of flying geese*

```
| pica point1 move shift|
point1 := 200@300.
move := 25@0.
shift := -25@50.
pica := Bot new.
5 timesRepeat:
 [10 timesRepeat:
 [pica
 triangleAt: point1
 delta1: 25@-25
 delta2: -25@25.
 point1 := point1 + move].
 point1 := point1 + shift.
 move := move negated.
 shift setX: shift x negated setY: shift y].
```

Script 21-16 uses the methods `negated` and `setX:setY:` within a double loop to generate the flying geese pattern over a large region of the screen. The inner loop is in the spirit of Script 21-12, except that the orientation of the triangle and that of the translation are rotated so that a line of triangles is now horizontal. The outside loop makes a translation of the last triangle to bring it over the line of triangles just drawn using the point variable `shift`; then, the translation is reversed so that the next line is drawn in reverse order. The triangles, however, are still drawn with the same orientation. The fact that a second line of triangles appears to point in the opposite direction is an optical illusion resulting from the fact that the two lines of triangles touch each other. Note that the variable `shift` must be transformed in a special way: the sign of its  $x$ -coordinate is reversed at the end of each line to compensate for the last translation, which is not drawn.

## Further Experiments

### Experiment 21-6 (Translating a Robot by a Point)

Defining methods with well-defined and simple behavior is a way to simplify your code, as explained in Chapter 16. How would you define a method called `translate: aPoint` that translates the receiver, a robot, by `aPoint` from its current position? Before looking at my solution in Method 21-1, try to find your own implementation.

**Method 21-1.** *Translating a robot by a point*

```
translate: aPoint
 "translate the receiver by aPoint"

 self goTo: (self center + aPoint)
```

Propose a different method, `translateX: x y: y`, which takes as argument the values for  $x$  and  $y$  separately.

### Experiment 21-7 (Using the Method `translate: 1`)

Change the definition of the method `triangleAt:point2:point3:` to use the method `translate: aPoint`.

### Experiment 21-8 (Using the Method `translate: 2`)

Using the method `translate: aPoint`, reimplement some of the methods you created in this chapter and compare their size and complexity.

## Summary

- A point is a pair of numbers: it has an  $x$ , or horizontal, coordinate and a  $y$ , or vertical, coordinate.
- `200@400` is a point whose  $x$ -coordinate is 200 and whose  $y$ -coordinate is 400.
- The Smalltalk coordinate system has its origin (0, 0) at the top left corner of the screen, and the  $y$ -axis has its positive direction downward, from the top of the screen to the bottom.
- To avoid trouble with points, surround them with parentheses when they are involved in complex operations.
- The methods `goTo: aPoint` and `jumpTo: aPoint` make the receiver move to the location of the argument, a point.



Here are some of the methods associated with points:

| <b>Message</b>               | <b>Description</b>                                                                                                                      | <b>Example</b>                        |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|
| <code>x @ y</code>           | Creates a point of given coordinates.                                                                                                   | <code>300 @ 600</code>                |
| <code>goTo: aPoint</code>    | Tells a robot to move to a given point.                                                                                                 | <code>pica goTo: 300 @ 600</code>     |
| <code>jumpTo: aPoint</code>  | Positions a robot at a given point.                                                                                                     | <code>pica jumpTo: 300 @ 600</code>   |
| <code>point1 + point2</code> | Creates a point whose coordinates are the sums of the coordinates of the two given points. This is useful for representing translation. | <code>(50 @ 200) + (300 @ 600)</code> |
| <code>point1 * number</code> | Creates a point whose coordinates are the products of the coordinates of the point and the number.                                      | <code>(50 @ 200) *3</code>            |
| <code>point1 negated</code>  | Constructs a point whose coordinates are the opposite of the original point.                                                            | <code>(50 @ 200) negated</code>       |
| <code>center</code>          | Returns the current position of a robot as a point.                                                                                     | <code>barPoint := pica center</code>  |



# Advanced Robot Behavior

**U**p to this point, I have presented only a subset of the messages that can be sent to a robot. In this chapter I present several more advanced messages that we will use in further experiments.

## Obtaining a Robot's Direction

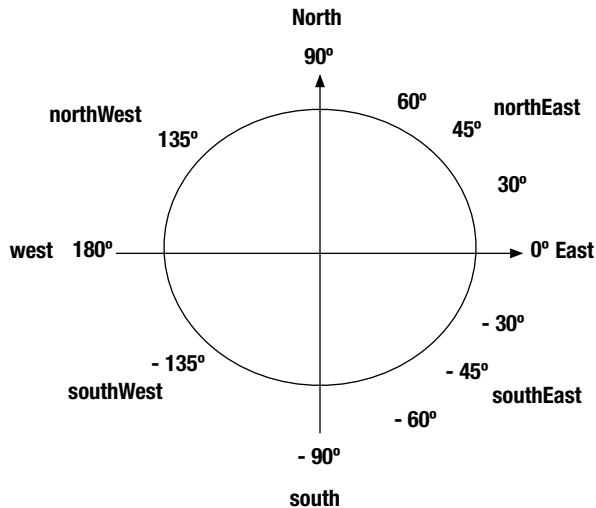
The first method that we need is the method `direction`, which returns the current direction of a robot in degrees. Using this message you can verify that the messages `south`, `north`, and so on that modify the direction of a robot in an absolute manner are in accord with their mathematical definitions, as shown in Script 22-1 and illustrated in Figure 22-1. Note that the direction is always a number between -179 and 180 degrees.

**Script 22-1.** *Illustrating the method `direction`*

```
| robot |
robot := Bot new
robot north.
robot direction.
-Printing the returned value: 90
robot west.
robot direction.
-Printing the returned value: 180
robot east.
-Printing the returned value: 0
robot southEast
-Printing the returned value: -45
```

## Pointing in a Direction

The message `turnTo: aDirection` tells the receiver to turn to a particular direction given as an angle in degrees. I have called the argument of this message *aDirection* rather than *anAngle* to stress the fact that the parameter represents an absolute angle, based on the mathematical definition, as illustrated in Figure 22-1. Thus for example, the message `turnTo: 45` turns the receiving robot to point northeast regardless of where it was pointing before. Contrast this with the message `turn: 45`, which turns the robot 45 degrees to the left *relative to its current position*. This means that the expression `robot turnTo: 90` is equivalent to any of `robot north`, `robot turnTo: 270`, and `robot turnTo: 360 + 90` (see Script 22-2). Note that the numbers in Figure 22-1 are the smallest in absolute value to represent an angle, and in fact, the implementation in Smalltalk always makes the robot turn through this smallest amount.



**Figure 22-1.** *The angles associated with absolute direction messages*

**Script 22-2.** *Illustrating the method* `turnTo: anAbsoluteAngleInDegrees`

```
| robot |
robot := Bot new.
robot turnTo: 90.
"leads to the same result as: robot north"
```

## Distance from a Point

The next piece of information that we would like to obtain from a robot is its distance from a given point. This is just what you get when you send the message `distanceFrom: aPoint` to a robot. Such information is useful, for example, when you want to know whether a robot is approaching a given location.

**Script 22-3.** *Illustrating* `distanceFrom: aPoint`

```
| robot |
robot := Bot new.
robot jumpTo: 100@100.
robot distanceFrom: (140@130)
-Printing the returned value: 50
```

## Back in the Center of the Screen

Another useful message is `home`, which places the receiving robot where it appeared when it was created, that is, in the center of the screen.

## Location If It Moved

Sometimes, we would like to know the location that a robot would be in if it were to move a certain distance in its current direction. We will make considerable use of this functionality when we simulate animal behavior. For this purpose, the method `positionIfGo: aDistance` is defined, and it can be used as shown in Script 22-4.

**Script 22-4.** *Illustrating* `positionIfGo: aDistance`

```
| robot |
robot := Bot new.
robot jumpTo: 100@100.
robot east.
robot positionIfGo: 100
-Printing the returned value: 200@100
```

The point at horizontal distance 100 pixels from the point 100@100 is the point 200@100.

## In a Box

Before reading the solution, try to define a method `go: anInteger ifStayInBox: aRectangle` that moves the receiver only if such a move would have it end up inside a specified rectangle, as shown in Script 22-5.

**Script 22-5.** *Using the method* `go: anInteger ifStayInBox: aRectangle`

```
Bot new
 go: 100
 ifStayInBox: (Rectangle center: World center extent: 400@300)
```

To create a rectangle, you can use, for example, the method `center:extent:`, which creates a rectangle centered at a point. The expression `(Rectangle center: World center extent: 400@300)` returns a rectangle whose center is the center of the screen and whose base and height are 400 and 300 pixels. You can ask a rectangle whether it contains a point using the method `containsPoint: aPoint`. To determine what the position of a robot would be if it moved forward a certain distance, you can use the method `positionIfGo: aDistance`.

Method 22-1 shows the definition of such a method `ifStayInBox`. It uses the method `positionIfGo:`, which computes the location of a robot if it were to move forward a given distance in its current direction. We will use the idea of constraining the movement of a robot to represent animal behavior in Chapter 23.

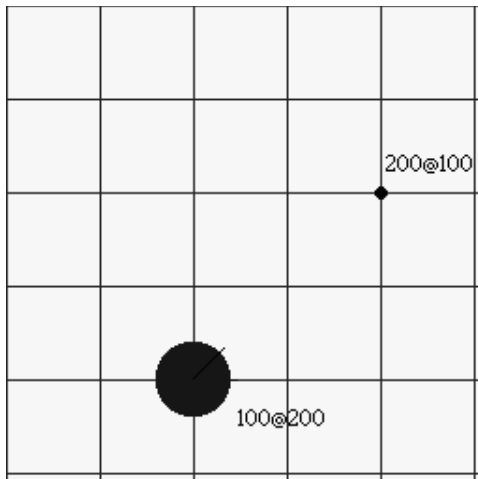
**Method 22-1.** *Move the receiver only if it will land inside a given rectangle.*

```
go: anInteger ifStayInBox: aRectangle
 "Move forward the receiver only if it stays within a Rectangle"

 (aRectangle containsPoint: (self positionIfGo: anInteger))
 ifTrue: [self go: anInteger.]
```

## Heading toward a Point

It is sometimes necessary to ask a robot to direct itself toward a certain point. The method `pointAt: aPoint` changes the direction of the receiver so that it points in the direction of the point `aPoint`. Once the method has been executed, the receiving robot points in the direction of the specified point, as shown in Figure 22-2. You can verify this using the method `direction`, as shown in Scripts 22-6 and 22-7.



**Figure 22-2.** *A robot points at the point 200@100 after being sent the message `pointAt`.*

**Script 22-6.** *Illustrating `pointAt: aPoint`*

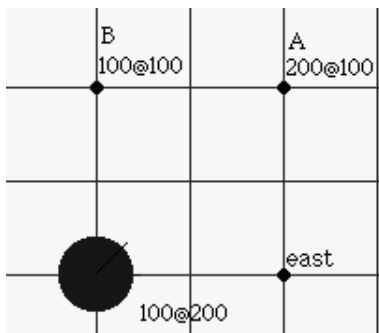
```
| robot |
robot := Bot new.
robot jumpTo: 100@200.
robot pointAt: 200@100.
robot direction
-Printing the returned value: 45
```

**Script 22-7.** *Illustrating pointAt: aPoint*

```
| robot |
robot := Bot new.
robot jumpTo: 100@200.
robot pointAt: 200@300.
robot direction
-Printing the returned value: -45
```

However, being able to point at a given location may not be enough. Sometimes, we would like to know the angle through which a robot should turn to point at a given location. The method `angleToPointAt: aPoint` returns the difference between the receiver's current direction and the direction that would point it toward `aPoint`.

Figure 22-3 and the Script 22-8 illustrate this method. In the figure, a robot is pointing in the direction of point A, and its direction is 45. Now the expression `robot angleToPointAt: 200@100` returns 0, because the robot is *already* pointing toward this point. Then the robot is turned to point at point B. Now the expression `robot angleToPointAt: 200@100` returns -45, because the robot would have to turn clockwise through 45 degrees to point at point A.



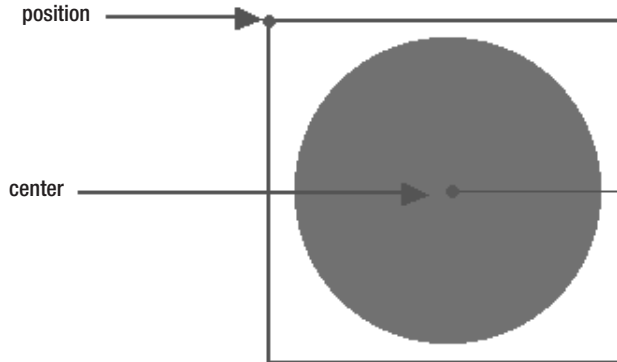
**Figure 22-3.** *The method `angleToPointAt: aPoint` returns the angle through which a robot would have to turn to point at `aPoint`.*

**Script 22-8.** *Illustrating angleToPointAt. See Figure 22-3.*

```
| robot |
robot := Bot new.
robot jumpTo: 100@200.
robot pointAt: 200@100.
robot direction.
-Printing the returned value: 45
robot angleToPointAt: 200@100.
-Printing the returned value: 0
robot pointAt: 100@100.
robot angleToPointAt: 200@100.
-Printing the returned value: -45
```

## Center versus Position

Finally, you may want to know the current location of a robot on the screen. To obtain this information you can use the method `center`, which returns the position of the robot's pen. A robot also understands the method `position`, which is provided by Squeak and returns the top left corner of the rectangle representing the robot, as shown in Figure 22-4. Normally, you do not have to use the `position` method, which is provided by Squeak itself and is not specific to the `Bot` class.



**Figure 22-4.** *The difference between the position and the center of a robot.*

## Summary

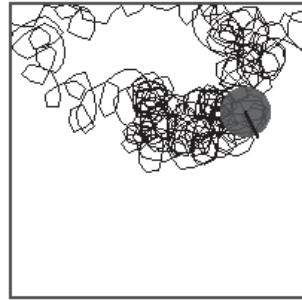
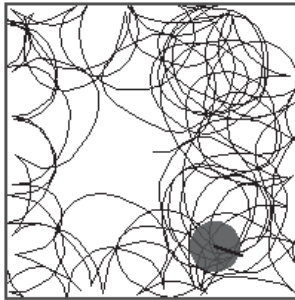
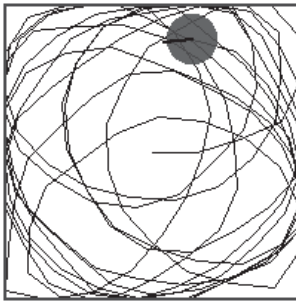
The following table summarizes the methods introduced in this chapter:

| Message                             | Description                                                                                                          | Example                                     |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| <code>angleToPointAt: aPoint</code> | Tells the receiver to compute the angle through which it would need to turn in order to point at <code>aPoint</code> | <code>berthe angleToPointAt: 100@100</code> |
| <code>turnTo: aDirection</code>     | Tells the receiver to turn to point in the direction <code>aDirection</code>                                         | <code>berthe turnTo: 90</code>              |
| <code>direction</code>              | Tells the receiver to report its current direction                                                                   | <code>berthe north; direction</code>        |
| <code>position</code>               | Tells the receiver to report its upper left point                                                                    | <code>berthe position</code>                |
| <code>center</code>                 | Tells the receiver to report the position of its pen                                                                 | <code>berthe center</code>                  |
| <code>beVisible</code>              | Tells the receiver to display itself                                                                                 | <code>berthe beVisible</code>               |
| <code>beInvisible</code>            | Tells the receiver to hide itself                                                                                    | <code>berthe beInvisible</code>             |
| <code>distanceFrom: aPoint</code>   | Tells the receiver to report its distance from <code>aPoint</code>                                                   | <code>berthe distanceFrom: 140@130</code>   |





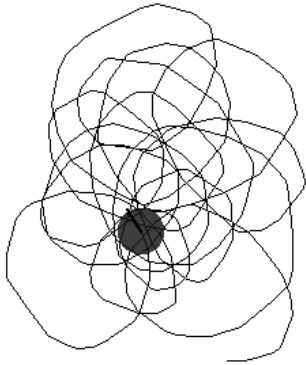
# Simulating Animal Behavior



**C**omputers are good for modeling the world in which we live, from plant growth to economic models to the behavior of markets. In this chapter I will show you how to model certain animal behaviors and use simulation to understand the factors that influence them. Together we will model strategies that animals develop to walk, escape, find food, and remain in a friendly environment.

## Wandering

Let's start by modeling how an animal might wander. The basic approach to simulating an animal's wandering behavior is to write a loop in which the animal walks and turns somewhat at random. We will use a random number in defining a method `wandering:` that makes the receiver wander by walking a random number of steps, turning through a random angle, and repeating these two moves `aNumber` times. (To obtain a random integer between 1 and 30, send the message `atRandom` to the number 30.) A possible result of executing the method is illustrated in Figure 23-1.



**Figure 23-1.** *An animal wanders by walking a random number of steps and then turning randomly.*

Method 23-1 presents one way to define the simple behavior described above and illustrated in the figure. Here we simply tell a robot to move randomly a distance between 1 and 30 pixels and turn left through an angle between 1 and 30 degrees. Script 23-1 shows how to invoke this method.

### Script 23-1

```
Bot new wandering: 500
```

### Method 23-1

```
wandering: n
```

```
"Make the robot walk a random distance and turn through a random angle n times"
```

```
n timesRepeat:
```

```
 [self go: 30 atRandom.
```

```
 self turnLeft: 30 atRandom]
```

Of course, animals do not wander at random. Eons of evolutionary development have created animals that move and turn in response to stimuli in their environment. Perhaps an animal turns and wanders until a certain event occurs. To begin to model such behavior in which the animal loops until some event happens, which we will model by having the user press a mouse button, you could use a conditional loop. The method `wanderingUntilButtonPressed` (Method 23-2) illustrates this point. It allows an animal to wander until the user presses one of the mouse buttons. Since the loop is executed extremely rapidly, it may happen that your computer seems blocked, so just keep holding down the button.

**Method 23-2.** *An animal wanders randomly until an event occurs.*

#### `wanderingUntilButtonPressed`

"Make the robot walk a random distance and turn through a random angle until a mouse button is pressed"

```
[self anyButtonPressed] whileFalse:
 [self go: 30 atRandom.
 self turnLeft: 30 atRandom]
```

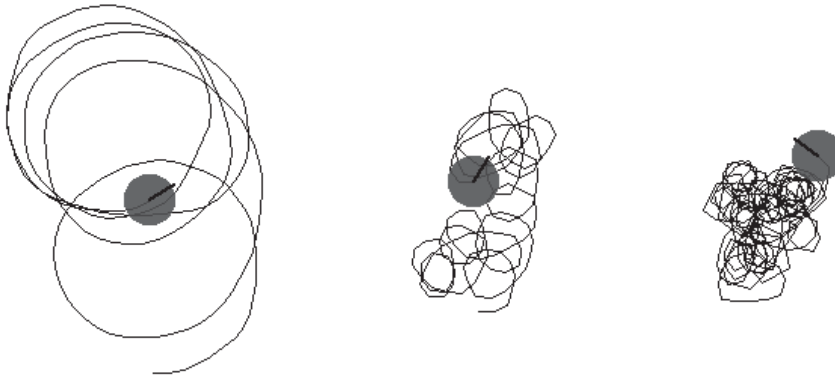
## Separating Influences

Method 23-1 is interesting, but it mixes two aspects of the animal's wandering: the random walking and the random changes of direction. Moreover, every time you want to try a new value of the angle or the number of steps, you have to recompile the method `wandering: n`. Therefore, define a method `wandering: n maxAngle: anAngle` that takes as its second argument the maximum value of the random angle through which the receiver should turn. In this method, let the robot always move forward a fixed distance. Such a method may be invoked as in Script 23-2.

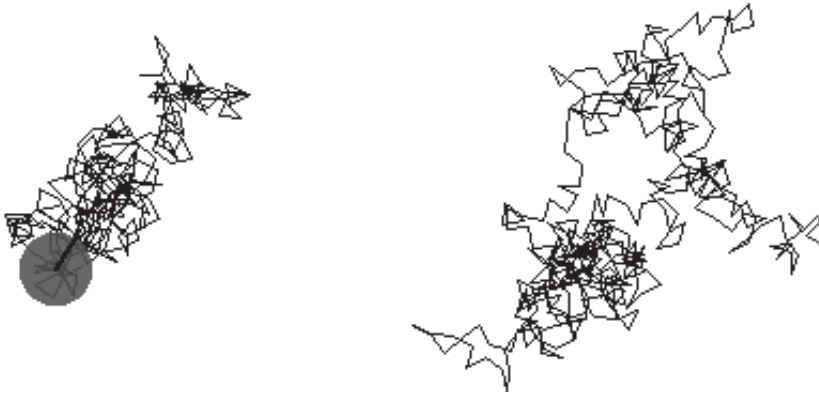
#### **Script 23-2**

```
Bot new wandering: 500 maxAngle: 60
```

Try to guess what the animal trace will look like before executing your scripts. Experiment with different angle values. Figures 23-2 and 23-3 show different results with 15, 60, 90, 180, and 360 degrees.



**Figure 23-2.** Walking with random angles of maximum 15 (left), 60 (center), and 90 (right) degrees



**Figure 23-3.** Walking with random angles of maximum 180 (left) and 360 (right) degrees

## Studying the Influence of the Length

We have just been examining the influence of the angle on the shape of the walk. What are your hypotheses about the influence of the length? What should happen if an animal walks a random distance and then turns through a constant angle?

## Studying the Influence of the Side to Which the Animal Turns

Up to now, we have always had our animal turn to the left. We can study the influence of the ability to turn to only one side or to two sides. Try to think of a solution to this programming problem before reading my solution. Hint: Note that `atRandom` returns a number between the receiver and 1. Therefore `2 * atRandom` returns either 1 or 2.

One possible way to generate a random choice is by introducing a random number (1 or 2) to represent the side chosen, as sketched in Script 23-3. Another idea is to generate a random number twice the maximum angle desired and subtract that angle from the random number. More precisely, to obtain a number between -45 and 45, you can generate a random number between 1 and 91 and subtract 46 from that number, as shown in Script 23-4. Figure 23-4 shows what can happen when these strategies are used, and you can see that the path looks more like that of a real animal, a snail, say, than the previous ones.



**Figure 23-4.** An animal wanders by walking and then turning randomly, where turning to the right and to the left are equally likely.

**Script 23-3.** A random number (1 or 2) determines whether the animal turns left or right.

```
...
 left := 2 * atRandom.
 left = 1
 ifTrue: [self turnLeft: ...]
 ifFalse: [self turnRight: ...]
...

```

**Script 23-4.** *The maximum turning angle is subtracted from a random number to yield an angle that can be positive or negative.*

```
...
rdAngle := ((1 + (angle * 2)) atRandom) - (1 + angle).
self turn: rdAngle.
...
```

## Trapped in a Box

Now I would like to constrain the wandering of the animal so that it remains inside a box. This will allow us to study different strategies that bugs seem to follow when they find themselves in such an uncomfortable situation. Doing this is easy. Before telling the animal to move through a certain distance, you have to check whether the location where it would end up is contained in the box. Such a constrained move has already been presented in chapter 22, with Method 22-1, but I will repeat the code in Method 23-3.

### Method 23-3

```
go: aDistance ifStayInBox: aRectangle
"Move the receiver forward only if it stays within aRectangle"
(aRectangle containsPoint: (self positionIfGo: aDistance))
ifTrue: [self go: aDistance]
```

To create a box, I can create a rectangle, as shown in Script 23-5. This script creates a box 200 pixels on a side around the current position of the animal.

**Script 23-5.** *Create a rectangle centered on an animal.*

```
| pica rectangle |
pica := Bot new.
rectangle := Rectangle center: pica center extent: 200@200.
```

To improve the visual effect of the simulation and if you want to see the box on the screen, you have to create a “rectangle morph,” that is, a graphical object whose shape is a rectangle, as described in Script 23-6. This script first creates a rectangle, then creates a rectangle morph whose bounds are those of the rectangle, with a transparent interior and blue border. Finally, the rectangle morph is displayed by invoking the method `openInWorld`.

**Script 23-6.** *Visualizing a rectangle centered on an animal*

```
| pica rectangle rm |
pica := Bot new.
rectangle := Rectangle center: pica center extent: 200@200.
rm:= RectangleMorph new.
rm bounds: rectangle.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blue.
rm openInWorld.
```

Now I will define the method `box: aRectangle` that draws a box representing the rectangle, as you will certainly get considerable use out of it.

**Method 23-4.** *Draw a visible box on the screen.*

**box: aRectangle**

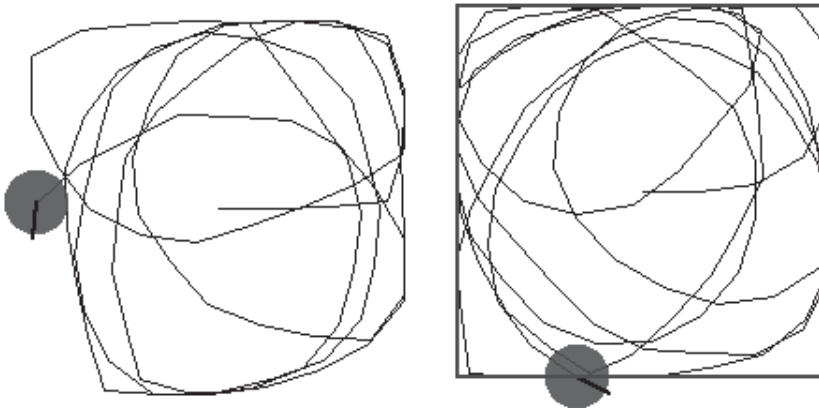
```
"Draw a morph to represent the rectangle"
| rm |
rm := RectangleMorph new.
rm bounds: aRectangle.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blue.
rm openInWorld
```

Now combine all the pieces to create an animal inside a box and test the method `go: anInteger ifStayInBox: aRectangle`. The next natural question is this: how can we more accurately model the behavior of the trapped animal? Imagine different alternatives. You should experiment with a number of variations.

## Following Borders

One approach is to make the animal turn a little bit when it can't move and try again. This is what Method 23-5 specifies. When the robot can move, it just wanders, but if its motion would cause it to bump into a box border, it turns through an angle. This behavior is contained in a loop that makes the robot turn and try again to walk. If the turn is not big enough, it just continues to turn until it can move again.

Scripts 23-6 and 23-7 together produce results similar to those shown in Figure 23-5. Script 23-6 displays a rectangle morph to materialize the box, and Script 23-7 controls the animal's motion.



**Figure 23-5.** *An animal trapped in a box turns until it can move without bumping into the border.*

**Script 23-7.** *An animal trapped in a box tries to move.*

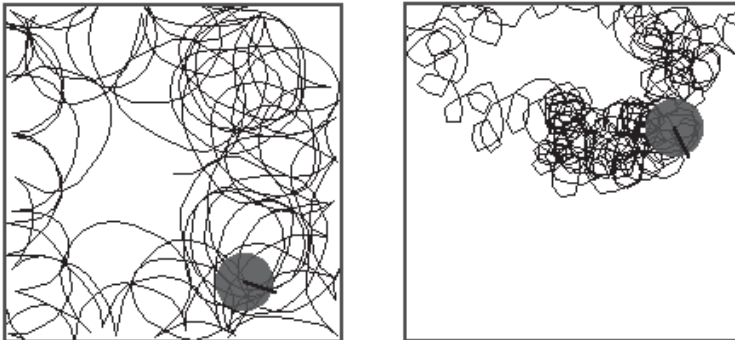
```
| t rec |
t := Bot new.
rec := (Rectangle center: t center extent: 200@200).
t follow: 500 borderOfBox: rec
```

**Method 23-5.** *The robot turns if it can't move and tries again.*

```
follow: n borderOfBox: aRectangle
self box: aRectangle.
n timesRepeat:
 [(aRectangle containsPoint: (self positionIfGo: 30))
 ifTrue: [self go: 30.
 self turnLeft: 30 atRandom]
 ifFalse: [self turnLeft: 1]]
```

## Flying to the Opposite Border

Another strategy to simulate is that of an insect that tries to fly to the opposite border. I will let you code this behavior, a possible trace of which is shown in the left pane of Figure 23-6.



**Figure 23-6.** *Left: The insect moves in the opposite direction when confronted by a wall. Right: The insect chooses a random direction when confronted by a wall. This picture was obtained by setting the maximum length of the random walk to 10 pixels and turning a maximum of 90 degrees. To escape, it will make a random turn of a maximum of 360 degrees.*

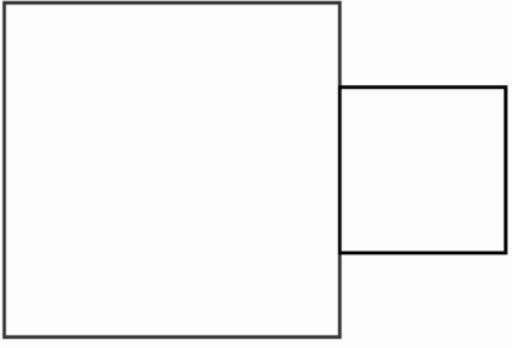
## Random Direction

Another choice is for the animal to change its direction at random. I will let you define this behavior, for which a possible trace is shown in the right pane of Figure 23-6.



## Introducing an Exit in the Box

Now you can add another rectangle to represent an exit in the box as illustrated in Script 23-8 and displayed in Figure 23-7.



**Figure 23-7.** A box with an exit

### Script 23-8. Visualizing a box containing an exit

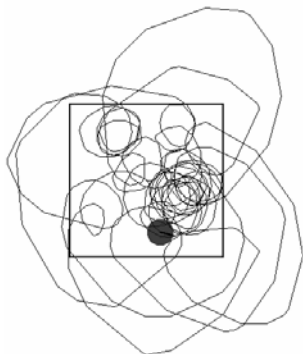
```
| box rm rm2 exit |
box := Rectangle center: World center extent: 200@200.
rm := RectangleMorph new.
rm bounds: box.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blue.
rm openInWorld.
exit := Rectangle origin: (box topRight + (2@50)) extent: 100@100.
rm2 := RectangleMorph new.
rm2 bounds: exit.
rm2 color: Color transparent.
rm2 setBorderWidth: 2 borderColor: Color black.
rm2 openInWorld.
```

Define a method to avoid having to repeat this code over and over. You might create a method named `escaping: aBox withExit: aExit` that checks first whether the next move would be contained in the rectangle representing the exit, and if this is the case, the method lets the animal escape.

## Staying in a Healthy Environment

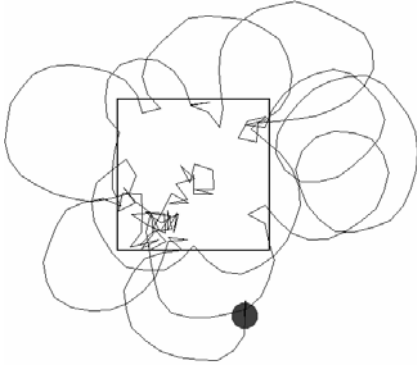
Have you ever wondered how slime bacteria manage to stay in moist places and avoid dry places? All right, perhaps you have not. In any case, I propose to simulate the strategies that certain basic life forms such as bacteria use to stay in an area where they have a greater chance of survival. I propose that you model two simple strategies discovered in the late 1950s using only changes in speed and direction. Note that when I talk about speed, I mean the distance that the animal can move at each step, which in our case will be the distance in pixels passed to the `go: method`.

The first strategy that the bacteria can follow is to change direction randomly in all circumstances but to change speed depending on the degree of environmental health. This strategy makes sense, because when a bacterium considers a region healthy, it will take more time for it to move a given distance, and hence it will stay longer in a healthy environment than in an unhealthy one. A possible path for a bacterium is shown in Figure 23-8.



**Figure 23-8.** *First strategy. The bacterium increases its speed when it finds itself in an unhealthy environment. It changes its direction at a constant rate. Here the speed in a healthy environment is a random number up to 25 pixels, while in an unhealthy environment it is 100. The interior of the rectangle represents the healthy environment.*

The second strategy is the opposite. The bacterium moves at constant speed but changes its direction depending on the health of the environment. It will change its direction on average by a larger angle when it finds a healthy place. Therefore, it has more chance to retrace its steps and remain within a smaller region. Certain very simple bacteria use this strategy to stay in an environment where they can find food. A possible path for a bacterium is shown in Figure 23-9.



**Figure 23-9.** *Second strategy. The bacterium increases the range of its directional change in a healthy environment. Here the speed is 25, and the change in direction is at most 36 in an unhealthy environment, and 360 in a healthy one.*

To implement these ideas you just have to imagine that the rectangle we have been using represents a healthy region. For the first strategy, you could define a method named `stayAtConstantAngleNTimes: aNumber in: aRegion`, as shown in Script 23-9. Experiment with different values of the angle through which the bacterium can turn.

**Script 23-9.**

```
| bacterium |
Bot clearWorld.
bacterium:= Bot new.
bacterium stayAtConstantAngleNTimes: 500
 in: (Rectangle center: bacterium center extent: 200@200)
```

Method 23-6 shows a possible solution.

**Method 23-6.** *A bacterium changes its speed to remain in a healthy environment.*

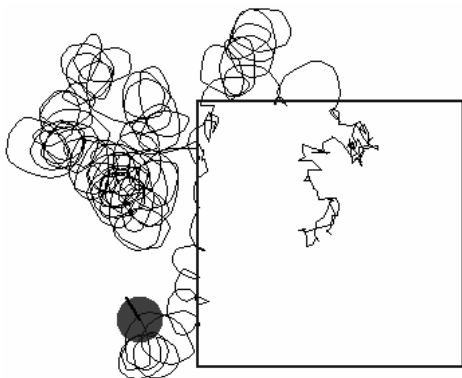
**stayAtConstantAngleNTimes: n in: aRectangle**

"The receiver tries to stay in a healthy environment by changing its speed, repeat n times."

```
self box: aRectangle.
n timesRepeat:
 [(aRectangle containsPoint: self center)
 ifTrue: [self go: 25 atRandom]
 ifFalse: [self go: 100 atRandom].
 self turn: 25]
```

## Further Experiments

We could imagine that the animal decreases its speed depending on the distance from the healthy zone. You might also introduce a bit of random behavior in the criterion (speed or turning angle) that otherwise does not change. You also could introduce the possibility for the bacterium to turn both clockwise and counterclockwise, as we already have implemented earlier in this chapter. As Figure 23-10 shows, the second strategy is not particularly efficient; the bacterium has no way of “knowing” whether it is moving in the direction of a good area, so it may end up staying outside of a healthy area for a long time and die as a consequence. Propose some other approaches to solve this aspect of the problem.



**Figure 23-10.** *Second strategy. Here the speed is a random number at most 10; the bacterium's change in direction is 36 degrees if the environment is unhealthy, and 360 if it is healthy.*

## Finding Food

Now I would like to model different ways that an animal could search for food. A first approach is based on the fact that an animal can locate its food visually. Again you can represent the food area by a rectangle.

### Comparing Distance

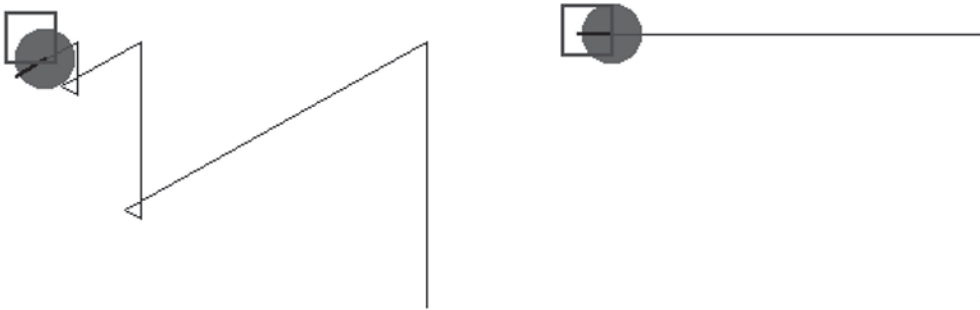
Imagine that an animal can evaluate the distance from a food source, which we will model as the animal's distance from the center of the rectangle used to represent the area in which food is located. Note that to obtain the distance between two points, you can use the method `dist::`. For example, `100@100 dist: 200@200` returns the distance between the points `100@100` and `200@200`. You can obtain the distance between a robot and a point using the method `distanceFrom: aPoint`.

Implement a method named, for example, `findFoodAreaByDistance: aFoodRectangle` that determines whether by walking one step forward the animal gets closer to the food area. When it is getting closer, it continues to move, but if it is getting farther away, it changes its direction by a certain fixed amount. Script 23-10 shows how the method might be used.

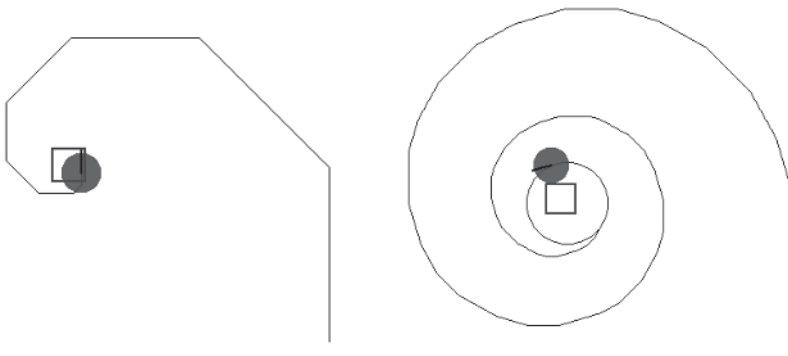
**Script 23-10.** Executing findFoodAreaByDistance:

```
| animal food |
animal := Bot new.
food := Rectangle center: 400@250 extent: 30@30.
animal findFoodArea: food
```

The behavior that I have described is somewhat naive, because there is no guarantee that turning through a certain angle will lead to an improved situation. Figures 23-11 and 23-12 present some traces. The right panel of Figure 23-12 shows a situation in which the animal comes within a certain distance of the food and then continually circles about the food rectangle without getting any closer. Try to figure out some solutions to this problem. Method 23-7 presents one possible definition of the method findFoodAreaByDistance:.



**Figure 23-11.** Finding food by comparing distances and simply turning. Left: turn 120 degrees. Right: turn 90 degrees.



**Figure 23-12.** Finding food by comparing distances and simply turning. Left: turn 45 degrees. Right: turn 15 degrees.

**Method 23-7.** *An animal approaches a food source by trying to decrease its distance from the food.*

**findFoodAreaByDistance: foodRectangle**

```
| food move |
self box: foodRectangle.
move := 10.
food := foodRectangle center.
[(foodRectangle containsPoint: self center)
 or: [self anyButtonPressed]] whileFalse:
 [((self positionIfGo: move) dist: food) > (self distanceFrom: food)
 ifTrue: [self turnLeft: 15]
 iffFalse: [self go: move]]
```

### Further Experiments

Here are some ideas for additional experiments. Change the position of the food or the direction of the animal at the beginning of its walk in Script 23-10.

Improve the behavior implemented by Method 23-7 so that once the animal realizes that it is moving in the wrong direction, it will not turn naively in a random direction but will check first to determine whether by turning one way it improves its situation and then changes direction accordingly.

To help you in your experimenting, do not hesitate to define new methods. For example, you could define a method `positionIfGo: aDistance andTurn: anAngle` that returns the position where the receiver would be if it were to turn through `anAngle` and move forward `aDistance` (see Method 23-8).

**Method 23-8.** *Find the position the receiver would be in if it turned through a certain angle and moved a certain distance.*

**positionIfGo: aDistance andTurn: anAngle**

```
"Return the position where the receiver would be if it turned
through anAngle and moved forward aDistance"
```

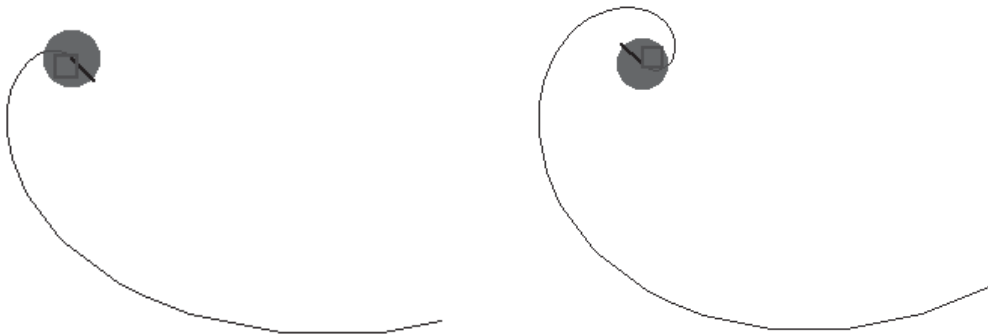
```
| position |
self turn: anAngle.
position := self positionInDirectionForDistance: aDistance.
self turn: anAngle negated.
^ position
```

We have been talking about the distance between the food and the animal, but it is unlikely that an animal has a way to make such precise measurements. Nevertheless, animals can estimate the distance from a food source in a variety of ways, such as the intensity of the smell of the food. For such an animal, reducing the distance to its food is equivalent to increasing the intensity of the smell.

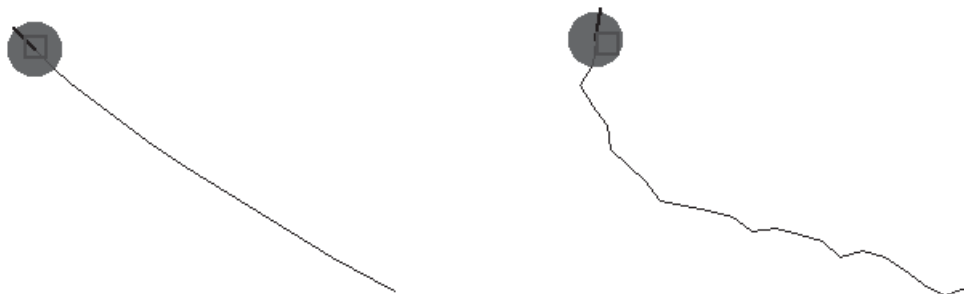
## Taking One's Bearings

In fact, a good way for an animal to be sure of reaching its food is to know exactly where the food is located. In such a case, the best strategy is to “keep your eyes on the prize” by constantly looking at the food and moving in its direction. Implement this approach using the method `pointAt:` and introduce some random movements to make the simulation a bit more realistic.

Now you can introduce the notion of speed and perturbation in the animal's trajectory. Define a method `keepABearing: aRectangle moving: aDistance turning: anAngle` that keeps the animal always pointing toward the center of the food area but moving and turning by a constant amount. Figures 23-13 and 23-14 show some results of this approach. With these constraints can you guess which is the more efficient way of reaching food: having a high speed and turning through a large angle or having a low speed and turning through a small angle? Of course, this simulation does not take into account that the food may move too.



**Figure 23-13.** Finding food by always pointing toward it. Left: speed 5 pixels turning 45 degrees. Right: speed 5 pixels turning 60 degrees.

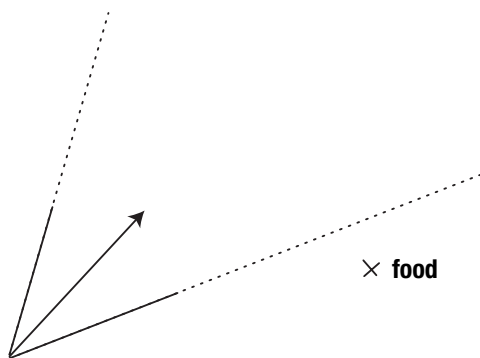


**Figure 23-14.** Finding food by always pointing toward it. Left: speed 15 pixels turning 5 degrees. Right: speed 5 pixels turning 60 degrees with random angle.

The left panel in Figure 23-14 shows that being able to point at the food with a small perturbation is one of the fastest approaches to reaching it. However, the speed has to be reasonable. Carry out some experiments with high speed to see whether this assertion is true. To obtain a more realistic simulation, introduce randomness in the angle and speed that could represent factors such as wind. For example, I introduced some randomness in the right panel in Figure 23-14. In addition to introducing randomness, implement the possibility that the angle may vary in both directions, as we have discussed earlier in this chapter.

## Simulating Vision

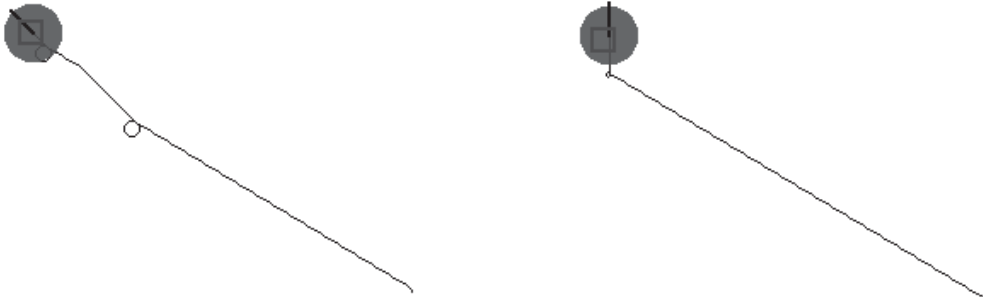
In the previous experiment, the animal could locate its food without any constraints. Now we would like to be a bit more realistic in our simulation. When an animal identifies its food by eye, it has a restricted angle of vision that prevents it from simply moving straight to the food. Imagine that our animal now has a single eye that is represented by an angle of vision within which the animal can see its food, as shown in Figure 23-15.



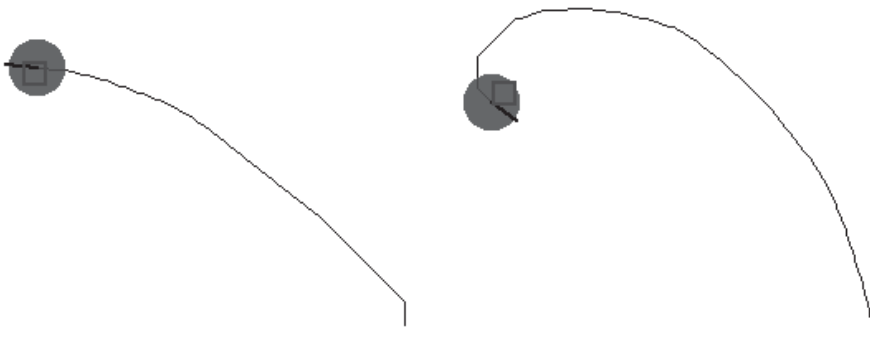
**Figure 23-15.** *The food lies outside the angle of vision, so the animal cannot see it.*

To define this behavior, I use the method `angleToPointAt:`, which returns the angle through which the animal should turn in order to point toward a given point. You could then decide, for example, that if the animal does not see any food, it turns around, and if it does see food, it moves straight in that direction. You need to be able to determine whether the angle through which the animal needs to turn is smaller than half of its view range. This is what the expression `(self angleToPointAt: aRectangle center) abs < (viewRange / 2)` does in Method 23-9. The expression `self angleToPointAt: aRectangle center) abs` returns the angle between the animal's current direction and its food. Now put all of these ideas together and define the method `lookAndFindFoodAt: aRectangle viewRange: viewRange turning: anAngle`, which implements this behavior. Method 23-9 provides a possible solution, but try to invent your own approach. Some traces are shown in Figures 23-16 and 23-17.



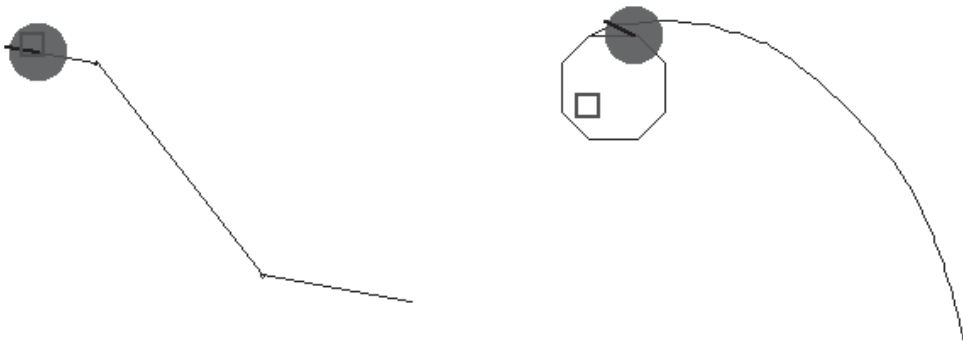


**Figure 23-16.** Finding food by turning only when the food is not seen. Left: vision angle 10 degrees, turning angle 15 degrees. Right: vision angle 40 degrees, turning angle 60 degrees.



**Figure 23-17.** Finding food by turning only when the food is not seen. Left: vision angle 15 degrees, turning angle 2 degrees. Right: vision angle 35 degrees, turning angle 3 degrees.

This approach is again rather naive, since it may loop (see Figure 23-18, right panel). Indeed, the change in angle does not necessarily lead to a better situation. I also suggest that you make the animal move continuously and not simply turn around when it does not see its food.



**Figure 23-18.** Finding food by turning only when the food is not seen. Left: vision angle 35 degrees, turning angle 80 degrees. Right: vision angle 45 degrees, turning angle 2 degrees.

**Method 23-9****lookAndFindFoodAt: aRectangle viewRange: viewRange turning: anAngle**

```
[(aRectangle containsPoint: self center)
 or: [self anyButtonPressed]] whileFalse:
 [(self angleToPointAt: aRectangle center) abs < (viewRange /2)
 ifTrue: [self go: 15]
 ifFalse: [self turn: anAngle]]
```

## Summary

The approaches to modeling animal behavior presented in this chapter are simple, but already they produce some interesting results. In general, the introduction of perturbation helps to simulate real behavior. For further projects, you might combine several behaviors and try to link certain inputs such as the angle of vision. Many other aspects of animal behavior can be modeled based on the examples presented in this chapter. I wish you lots of fun.

PART 5



# Other Squeak Worlds



# A Tour of eToy

This chapter presents a small tour of the eToy system. The eToy system provides an interface for manipulating objects, sending them messages, composing graphical scripts, and executing them. It is used in school by children ages 9 to 12. There is also a recent book, *Powerful Ideas in the Classroom*,<sup>1</sup> that presents in detail how to use eToy to teach mathematics and science. You can find a great deal of information related to eToy at <http://www.squeakland.org>.

In this chapter I will show you how to steer an airplane with a joystick, how to create animation, how to steer a car, and finally how to program a car to follow a road automatically. For all these tasks you should open a “morphic” project using the **World** menu (**open item**). If you use the BotsInc. environment, you have to go through some simple manipulation to obtain the full squeak menu. Select the **help...** menu item and then **reinstall the full menu**.

Now open a morphic project by again opening the **World** menu and choosing **open...** followed by **morphic project**. You should get a small window. Click on it to enter the project. You will be able to return to the current place using the menu item **previous project** of the **World** menu. Once inside the new project, you should select **flaps...** and then install the default shared flaps. It should take a moment, but you should get some new flaps, in particular those named “widgets” and “supplies” at the bottom of the screen.

If you use Squeak directly, select **open...** and then choose the **morphic project** menu item from the **World** menu and enter the new window that appears by clicking in it. Give your project a name and click on it to enter.

---

1. B. J. Allen Conn and Kim Rose, *Powerful Ideas in the Classroom: Using Squeak to Enhance Math and Science Learning* (Viewpoints Research Institute, 2003).

## Steering an Airplane

To steer an airplane, you first have to create an airplane. Then you have to obtain a joystick and create a script that connects the airplane with the joystick.

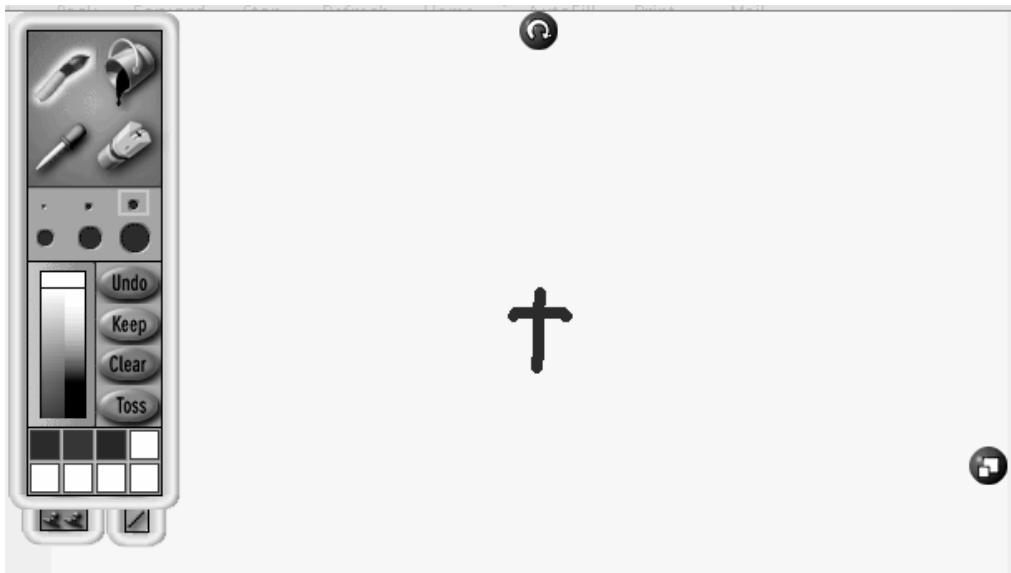
### Step 1: Drawing an Airplane

To obtain an airplane, the best thing to do is to draw it yourself. Open the blue flap called “widgets” (Figure 24-1) and drag the palette or paint editor from the flap to the desktop. This action opens a tool called “Paint” for painting and drawing on the screen.



**Figure 24-1.** Open the widgets flap to obtain the Paint tool.

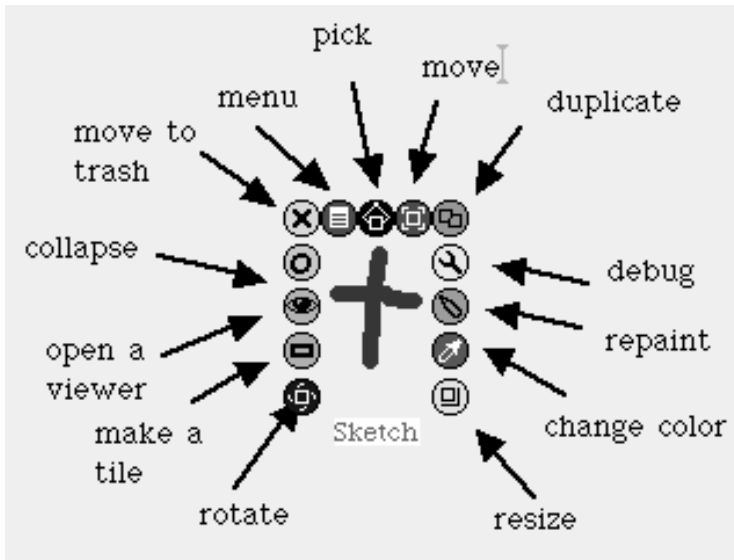
Using Paint, draw a small airplane, as shown in Figure 24-2. My airplane looks like a small cross, but you are welcome to draw a more sophisticated one. Once you are done with your drawing, press the button labeled “Keep.” The paint editor will disappear, and you will be left with a sketch that looks like an airplane. The airplane that you have drawn is called a *Player* in eToy jargon.



**Figure 24-2.** Drawing an airplane with the Paint tool

## Step 2: Playing with the Halo

Now if you click on your sketch with the right mouse button (or equivalent), you will see a halo appear around the sketch, as shown in Figure 24-3. Each halo handle has a color, a logo, and a function. Here I will explain each of them briefly. Note that different types of sketches will bring up a different set of handles. You can always get a description of a handle by letting the mouse hover over it for a second.



**Figure 24-3.** A halo surrounds a sketch.

- The pink handle with a cross symbol destroys the sketch. Depending on your preferences, the sketch is simply put in the trash or else completely destroyed. When it is put in the trash, you can retrieve and reuse it.
- The red handle with small rectangles brings up the menu associated with the sketch. The menu depends on the sketch you are interacting with. It provides many actions for changing the sketch.
- The black handle with pincers selects the sketch. Note that selecting a sketch changes its *container*. Use the brown handle to move the sketch inside its container. Use the red handle to embed the sketch in the sketch underneath it. The idea is that you can embed a morph into the morph under it by opening the red handle's menu and selecting the **embed into** menu item. This operation offers you a morph into which you can embed your morph. Once this is done, moving the embedding morph will move your morph too. With the black handle you can “de-embed your morph” from its container, while with the brown handle you only move your morph inside its container.
- The brown handle with a square moves the sketch without changing its container.
- The green handle with two squares duplicates the sketch.

- The grey handle with a wrenchlike tool offers debugging facilities, which are normally used only by expert squeakers.
- The dark grey handle has a pen with which you can repaint a sketch.
- The dark pink handle with an eyedropper changes the color of a sketch. However, it does not work with sketches such as our airplane. To change the color of a sketch that you have created, you need to use the dark grey “repaint” handle. If you select this handle, you will be taken back into the paint tool, where you can make any paint/color changes you wish. The dark pink handle is more appropriate to use for changing font colors, rectangle colors, and border colors.
- The yellow handle with a square and bar changes the size of the sketch.
- The blue handle with a small spinning square rotates the sketch.
- The orange handle with a small rectangle produces a tile that represents the object for the tiling system of eToy (more on this later).
- The cyan handle with an eye opens a viewer on the sketch. The viewer presents a graphical representation of the methods and instance variables of the object.
- The pale green handle with a circle collapses a sketch.

To program in the eToy environment we mainly use a viewer. Therefore, open a viewer on the airplane by clicking on the cyan handle with an eye. You should obtain the tools shown in Figure 24-4.



Figure 24-4. Opening a viewer

Figure 24-5 explains the main parts of a viewer. At the top, the name of the sketch can be modified. By default your sketch is named Sketch. I suggest that you call it “airplane” by editing the name in the viewer. In addition to changing the name, you can open a menu, as shown in the figure. With this menu you can add new variables to the object, obtain an empty script, or get a tile representing the object.

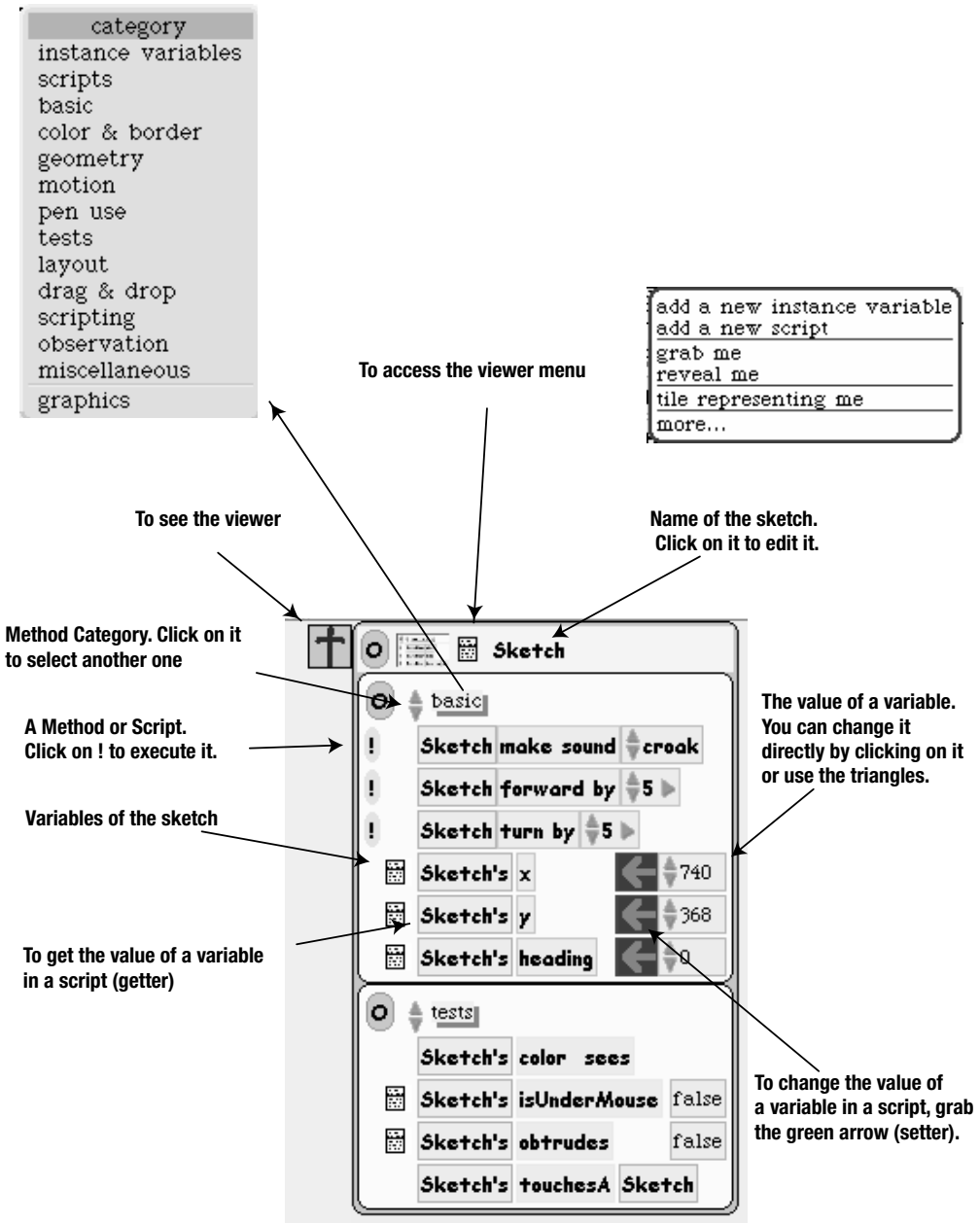


Figure 24-5. Understanding a viewer



In the viewer, categories are displayed. The name of a category is displayed to the right of the two up/down green triangles. In Figure 24-5, the categories *basic* and *tests* are displayed. You can navigate them using the small double triangles. You can also click on the category name itself to obtain a list of the categories.

Below each category can be seen a list of methods. When a method can be executed, there is a yellow exclamation mark in front of it. Press the exclamation mark in front of the method forward by and you will see that the airplane is moving forward. There are also methods that only access a variable or change a variable's value. We call these methods *accessors*: they are *getter methods*, which access values, and *setter methods*, which modify them.

Note that you can modify the value of a variable directly by using the green triangles or by typing a value directly in the variable box. Try, for example, to change the *x* variable to have the value 800. In a script, since you cannot interactively change the value of a variable, you should use the getter and setter methods. You can obtain the getter method for the variable *x* by dragging the *x* box onto the desktop. To get the setter method, you should drag the big green arrow, as shown in Figure 24-6.

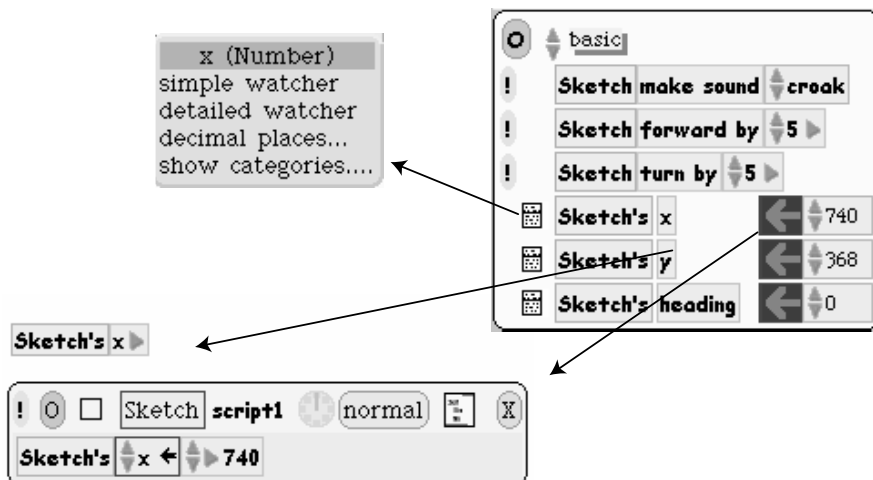


Figure 24-6. Accessing variables within a script

The number to the right of the big green arrow is the value of the variable. If you move the airplane using the brown or black handle, you will see the variables *x* and *y* changing their values.

You can also have *watchers*, which provide ways of spying on the values of variables. To create a watcher, you should click on the small menu icon that is to the left of a sketch variable. You should obtain the menu displayed in Figure 24-7. Then you can choose to create a simple watcher, which will display only the value of the variable, or a detailed watcher, which you can also use to change the value of the spied-on variable.

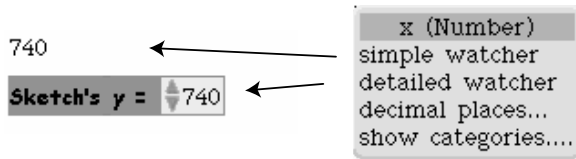


Figure 24-7. The watchers: spying on your variables

### Step 3: Dragging and Dropping a Method to Create New Scripts

If you take a method such as `forward` by from the viewer and drop it onto the desktop, you create a script. You should get, for example, the script shown in Figure 24-8.



Figure 24-8. A script is created.

To execute a script you have simply to click on the clock icon. The status of the clock is then set to *ticking* (see Figure 24-9), which means that your script is executed at regular time intervals.



Figure 24-9. When the clock is ticking, your script is being executed.

Figure 24-10 shows the different parts of a script: you can see the name of the sketch, the name of the script, a clock for starting and stopping script execution, a list of events to which the script can be linked, and the creation of a conditional, or test.

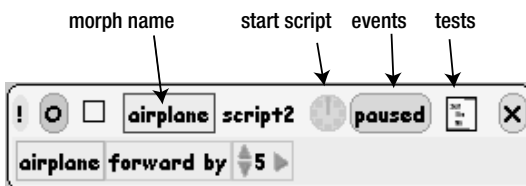


Figure 24-10. The different parts of a script

## Step 4: Adding Methods

When you execute the script, you can see that the airplane is flying in a straight line. No doubt you would like your airplane to be able to turn. To accomplish this, you need to add methods to the script. Drag the method `turn by` from the viewer and drop it into the script. The script should show you the place where you can drop the method by creating some green boxes. Drop the method into one of the boxes. You should get a script that looks like the one in Figure 24-11.

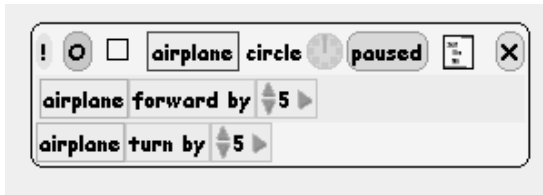


Figure 24-11. The method `turn by` has been added to the script.

Now when you execute the script, the airplane should turn in a circle. In fact, a sketch is similar to a robot, and you can see what the airplane is doing by telling it to lower its pen. To do so, look for the variable `penDown` in the viewer and put its value to `true`, as shown by the script displayed in Figure 24-12. The result is shown in Figure 24-13.

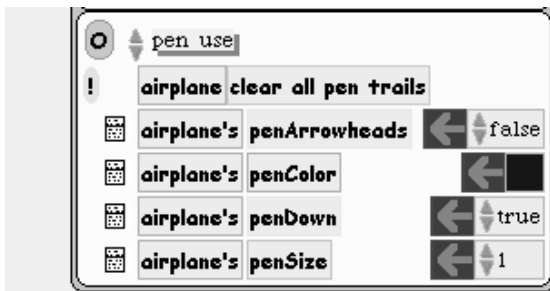


Figure 24-12. Once the pen is down, the airplane will draw a trace on the desktop, as shown in Figure 24-13.

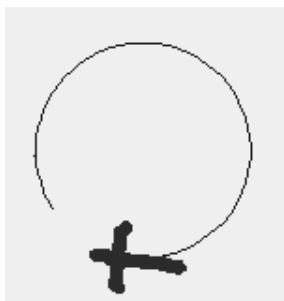


Figure 24-13. The airplane has put down its pen, and is seen to be flying in a circle.

Note that you can select when the script will be executed using the “events” button of the script. Figure 24-14 shows all the events that you can use. You can obtain this menu by clicking on the text to the right of the clock.

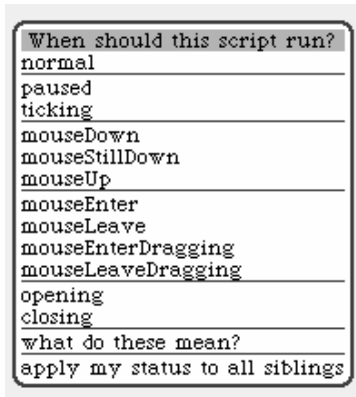


Figure 24-14. A list of available events

## Joysticks in Action

Flying in a circle is fun, but you would probably like to steer your airplane. Instead of the airplane always turning the same amount, you can vary the angle of turning depending on the state of a joystick.

### Step 1: Creating a Joystick

First create a joystick by dragging one from the “supplies” flap and dropping it onto the desktop. The red circle represents the top of the joystick (Figure 24-15), and with it you can indicate a direction and an amount of energy to be put into the movement.

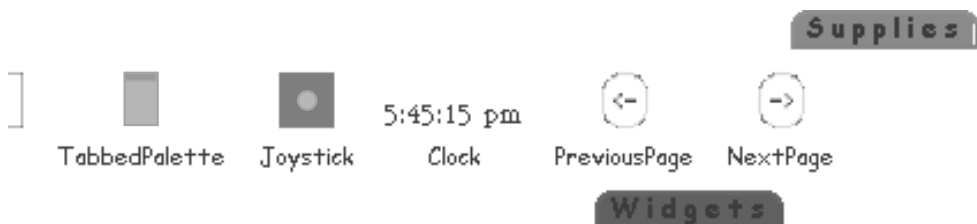


Figure 24-15. Creating a joystick

## Step 2: Experimenting with a Joystick

Second, open a viewer on the joystick using the turquoise handle. Then browsing the methods, you will find some useful methods under the category “joystick” (Figure 24-16). Move the joystick and look at the variables. The variable `amount` represents the amount of energy put into the movement; that is, you can choose not only the direction in which the joystick is moved, but the strength of the movement as well. The variable `angle` represents the direction in which you are moving the joystick, and I will let you guess the roles of the other two variables.

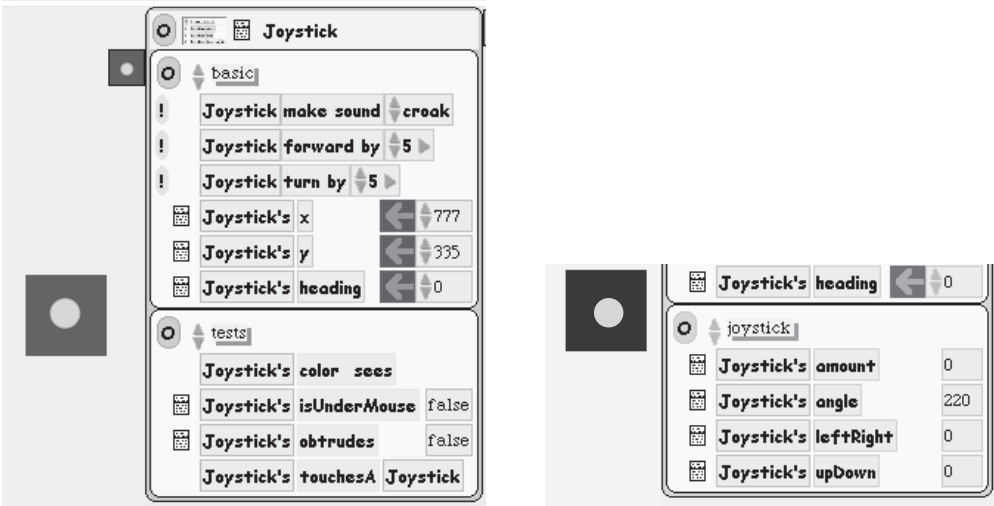


Figure 24-16. Methods under the category `joystick`

## Step 3: Linking the Joystick and the Script

In order to steer the airplane, you have to change the value of the method `turn by` in the script by a value given by the joystick. For this purpose, the `leftRight` variable appears to suit our needs. Drag the variable directly onto the argument of the `turn by` method in the script. It may take some time for the script to accept the variable, but you should obtain the script shown in Figure 24-17.



Figure 24-17. Now the joystick's `leftRight` variable will control the turning of the airplane.

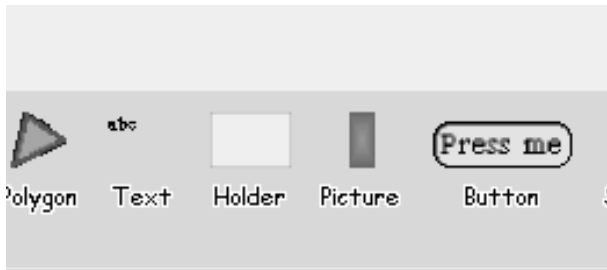
Now click on the clock and steer your airplane with the joystick. As you can see, steering the airplane could be improved by controlling how fast the airplane is steered. Try to find a solution. The amount variable can be of help.

## Creating an Animation

The idea for creating an animation is the following: first you draw individual animation frames and put them into an animation holder. Then you create a simple sketch whose graphical representation will be replaced by the animation elements. For this purpose you can write a script that makes the sketch look like the different steps that are contained in the holder.

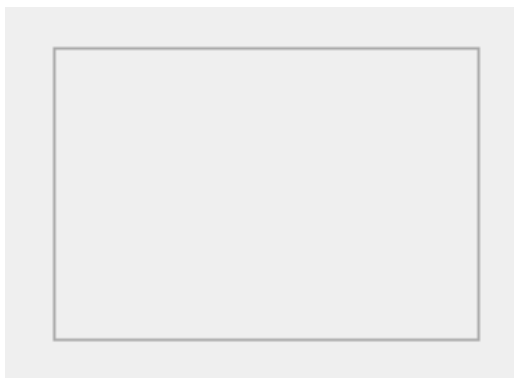
### Step 1: Creating the Holder

To create an animation, you need to create a holder. A holder is a graphical object that can contain other graphical objects. It also knows which is the currently selected item among the items that it contains. To create a holder, drag it from the red flap named “supplies” (Figure 24-18) and drop it onto the desktop.



**Figure 24-18.** The red flap can be used to create a holder.

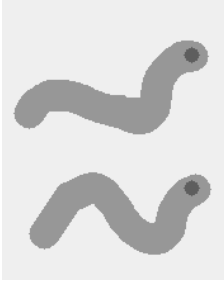
An empty holder is then created (Figure 24-19).



**Figure 24-19.** An empty holder

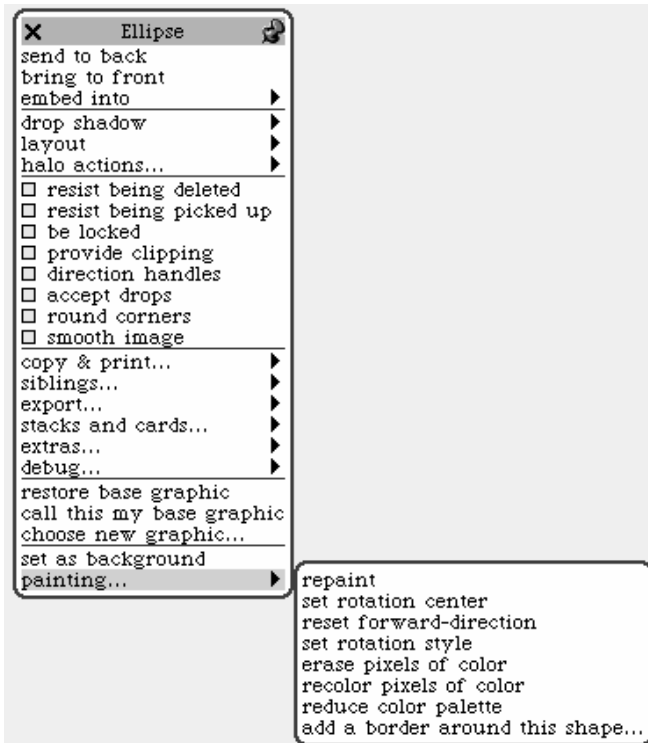
## Step 2: Drawing Animation Elements

For the second step, you should draw the picture that you want to animate using the paint editor presented earlier. I recommend that you first draw a picture, then duplicate it using the green handle. Then select the dark grey handle to repaint the sketch. This way, you can create several pictures by modifying one picture step by step. I am going to paint a worm in two different positions (Figure 24-20).



**Figure 24-20.** *A worm in two different positions*

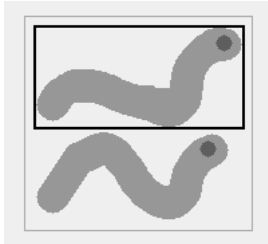
Note that you can also use the red handle painting item (see Figure 24-21), but I suggest that you use the handles provided as much as possible.



**Figure 24-21.** *The red handle painting menu*

### Step 3: Dropping the Pictures into the Holder

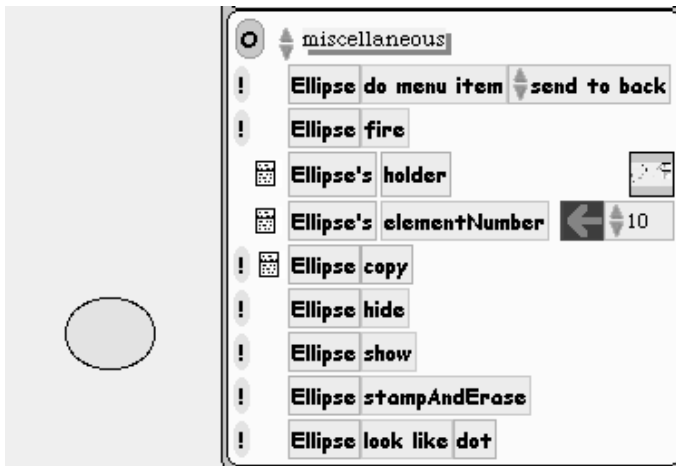
The next step is simply to drop the resulting pictures into the holder. A black rectangle (Figure 24-22) represents the currently selected picture in the holder.



**Figure 24-22.** *The worm in the black rectangle is the image currently selected.*

### Step 4: Creating a Simple Sketch Recipient of the Animation

Now you need a sketch whose graphical aspect will be used as a placeholder for the animation. Therefore, you should create a simple sketch such as an ellipse, which you drag and drop from the supplies flap. Then open a viewer on this sketch, as shown in Figure 24-23.



**Figure 24-23.** *A viewer on the sketch recipient*



## Step 5: Creating a Script with lookLike

From the simple sketch, here the ellipse, you can create a new script by dragging the method `lookLike dot` (shown in Figure 24-23) from the viewer to the desktop. This action should create the script described in Figure 24-24.

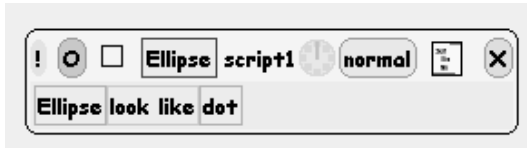


Figure 24-24. A script with `lookLike dot`

## Step 6: Displaying the Selected Animation Element

Now you should indicate that the ellipse should look like the currently selected element of the holder. Select the holder and open it in a viewer, as shown in Figure 24-25.

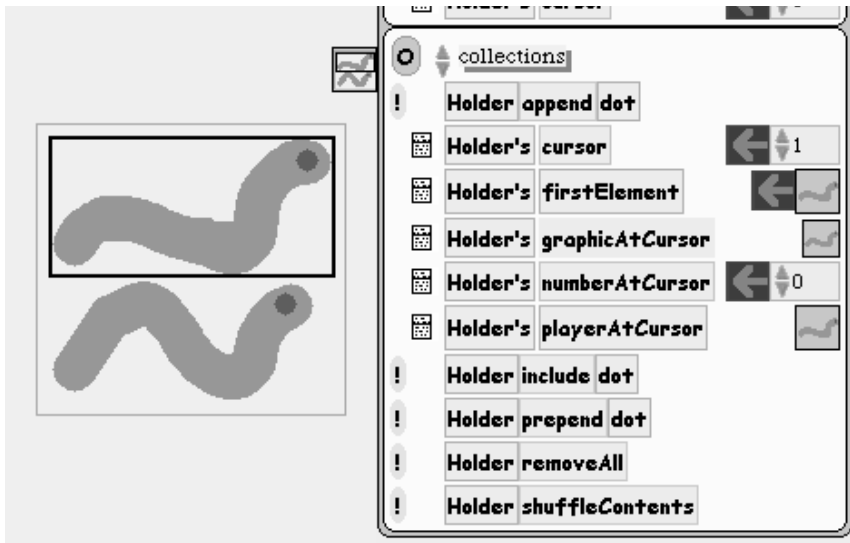
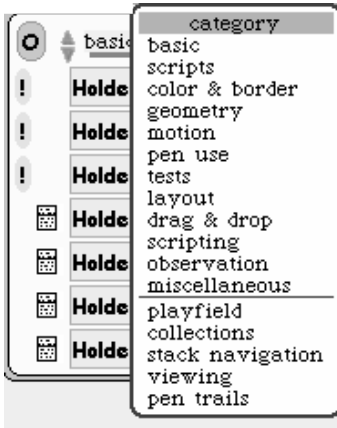


Figure 24-25. The holder opened in a viewer

Look for the category collection by choosing “basic,” as shown in Figure 24-26.



**Figure 24-26.** Pressing the “basic” button allows you to access collection.

From the list of methods drag the method named `playerAtCursor`, which returns the player, that is, the graphical element contained in the holder that is at the current position. Then drop it onto the square with a dot in the middle (this box represents an argument of the method `lookLike`). You will obtain the script shown in Figure 24-27.



**Figure 24-27.** Now the ellipse should look like the element pointed to by the cursor.

## Step 7: Changing the Currently Selected Element of a Holder

If you execute the previous script using the ticking button, the script will change the shape of the ellipse to the currently selected graphical element. We do not yet have an animation. For that, we need to find a way to change the currently selected item of a holder. In fact, a holder has an index, named `cursor`, that represents the rank of the currently selected item. It suffices to increment such an index to make the holder select another graphical element.

To change the value of the variable named `cursor`, drag the arrow (shown on the right in Figure 24-28) onto the following line of the previous script. You will obtain the script shown in Figure 24-29, which says that the variable contains the value 1.



Figure 24-28. *The cursor in the player*



Figure 24-29. *Introducing an assignment*

Now if you click on the double green triangles of the setter in the script, you will see that you have the possibility to increase the cursor by a given amount, as shown in Figure 24-30. Now your animation should work.



Figure 24-30. *Increasing the cursor*

## Another Way

To achieve the same effect, you can also mimic the expression `cursor := cursor + 1`, which will move the cursor to the next element. Therefore, drag the left part of the cursor from the viewer at the place of the 1 in the script, and you will get the script shown in Figure 4-31.



Figure 24-31. *Using the expression `cursor := cursor + 1`.*

Now you just have to click on the small green triangle at the end of the cursor box in the script to make the expression `+ 1` appear, as shown in this final script. As you can see, the first solution is easier.

I recommend that readers interested in using eToy in the classroom read the book mentioned at the start of this chapter; it presents many important pedagogical aspects that can be used with this example.

## Cars and Drivers

Now I would like to show you how you can build a script to control a car with a steering wheel. This exercise can be used by teachers to show the use of division to act as a demultiplier, as exists in real cars, but it is also simply fun to do.

### Step 1: Draw a Car and a Steering Wheel

Using the painter editor, draw a car and a steering wheel, as shown in Figure 24-32.



Figure 24-32. Draw a car and a wheel.

### Step 2: Turning the Car in a Circle

Open a viewer on the car, rename the sketch “car,” and then drag and drop the method forward as you did previously. The script shown in Figure 4-33 should be created.

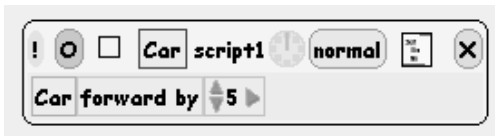


Figure 24-33. Drag and drop the method forward.

Then drag and drop the method turn below the previous method in the newly created script. You should obtain the script in Figure 4-34, which when executed makes the car go in a circle.



Figure 24-34. Drag and drop the method turn.

### Step 3: Using the Wheel's Heading

Now you have to link the angle through which the car turns with the wheel's position. We need to find a way to rotate the wheel and determine the amount of rotation. The first problem is easy to solve: bring up the halo, click on the blue handle, and turn it (Figure 24-35). This rotates the wheel. To solve the second problem, open a viewer on the wheel, rename it "wheel," and look for the heading variable in the viewer (the expression will be `wheel's heading`). When you rotate the wheel using the handle, the value of the variable changes. This variable represents the rotation angle compared to the original picture.



Figure 24-35. The wheel with its halo of handles

Now that you have all the pieces of the puzzle, you have to indicate that the car should turn not by a fixed amount but from the wheel's heading. To do this, drag the expression `wheel's heading` from the viewer onto the number 5 argument of the turn method in the previous script. You should now have the script shown in Figure 24-36.



Figure 24-36. The car is now turned from the wheel's heading.

Now if you click on the small clock in the script, it runs, and using the wheel's blue handle you can control the car. However, you will notice that this is not quite perfect. Teachers should elicit hypotheses from their students about the nature of the problem and possible solutions.

The problem is that the wheel's heading value should be divided so that the user can have finer control over turning. To do this, click on the rightmost small green triangle of the wheel's heading block. This adds extra boxes. Click on them to select division `//`. You have now a script that looks like Figure 24-37.



**Figure 24-37.** *Dividing the wheel's heading value yields finer control.*

The car, its steering wheel, and the halos all appear in Figure 24-38. Have fun controlling your car!



**Figure 24-38.** *Have fun with your car.*

Now we would like to program the car so that it finds its way along a road automatically. The idea is to equip the car with some sensors that tell it whether it is going off the road. Once the car has been equipped with sensors and properly programmed, we will put it on a road and see how it does in following it.

I will not show you a working solution to this problem because I think that you will enjoy experimenting with this one on your own. More to the point, my solution does not work very well.

## Step 1: Sensors

In eToy you can ask whether a certain color in a sketch can see another color underneath it. This capability can be used in constructing a sensor. A sensor will then be a simple dot of one color that will tell the car whether it is passing over another color. To equip your car with sensors, repaint the car and add two dots of colors, as shown in Figure 24-39.



**Figure 24-39.** *Two sensors have been added to the car.*

## Step 2: The Road

Using the paint editor, create a road of a single color. Later, you can try using multiple bands of colors.



**Figure 24-40.** *The road*

### Step 3: Conditions and Tests in eToy

Open a viewer on the car then drag and drop the method `forward by` onto the desktop. This creates a script like those that you are now used to creating.

Now you need to have a way to express different types of behavior depending on the value of the sensor. For that purpose you need to be able to express a condition, or test. To obtain a test, you should drag the small pale square box (second box from the right at the top of the script, near the cross button), as shown in Figure 24-41. Drag and drop this test tile into the script. You should now get a script that looks like the one in the figure.

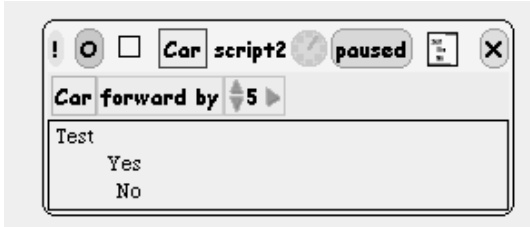


Figure 24-41. Adding a test to the script

Now you need to find a way to make your sensors active. Locate the method `color sees` in the category `test`, as shown in Figure 24-42. This method returns `true` or `false` depending on whether a colored part of the sketch of a given color passes over another given color.

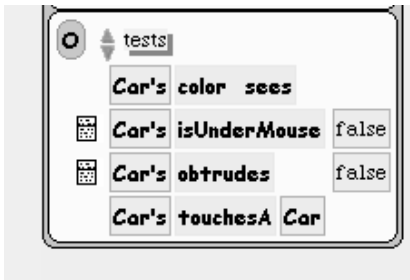


Figure 24-42. Adding a test to the script



Drag a `color sees` method into the script next to the word `Test`. You should obtain a script like the one in Figure 24-43.

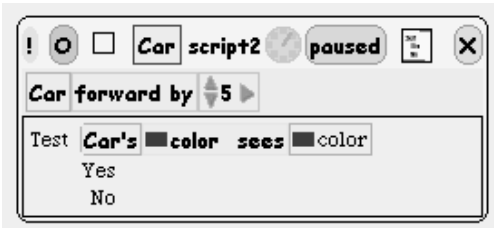


Figure 24-43. Adding a `color sees` method to the script.

## Step 4: Customizing Color-Based Tests

Now you need a way to define the color that the test should use. This is easy. You have simply to click on the colored square inside the method `color sees` itself. This automatically opens a color picker (see Figure 24-44). With the color picker, you can pick the color of one sensor. The first color of the script should change to reflect the color you selected. Do the same operation for the second colored box, but this time pick the color of the road (Figure 24-45).

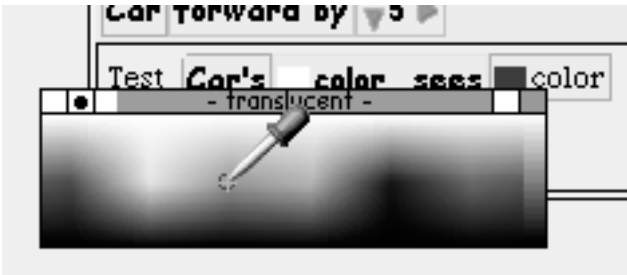


Figure 24-44. Using the color picker

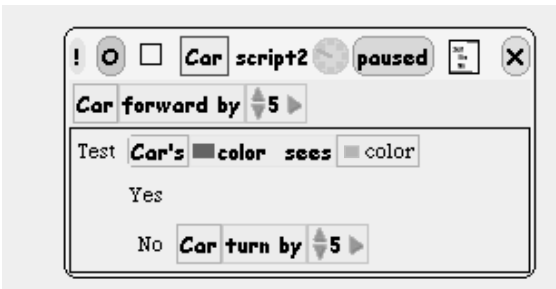


Figure 24-45. Choose the color of the road for what the car's color sees.

## Step 5: Adding Actions

Now you can specify that the car should turn when the sensor does not see the color of the road. You just have to drag and drop the method `turn` by beside the word `No` in the script. Now you can select the car, put it on the road, and press the clock to run the script. See Figure 24-46.



Figure 24-46. *The car runs along the road!*

Clearly, the car's behavior is not perfect, and I leave it to you to change it or find your own solution.

## Some Tricks

I would like now to show you some aspects of eToy that may help you. The eToy environment can be customized, so open the menu **playfield options...** (Figure 24-47) and let your mouse hover over the different items and read the balloon help. Here are some of the things you can do:

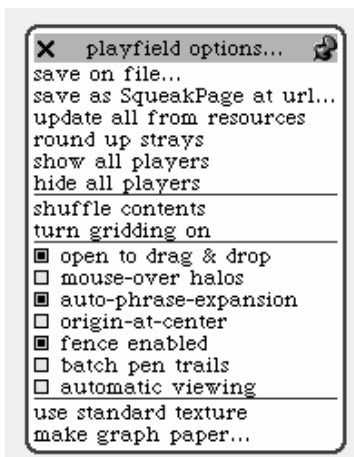


Figure 24-47. *The playfield options menu*

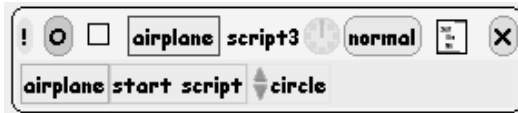
## Running Several Scripts

If you create several scripts, they will run in parallel. To control all the scripts available on the desktop, you can use the widget named “All Scripts,” which is available in the **widget** flap. Once you drop this widget onto the desktop, you get a panel (Figure 24-48) that allows you to run and stop all the scripts currently on the desktop.



**Figure 24-48.** *The script control panel*

Note that you can also use the method `start script` and the related method scripting contained in the category to start, pause, or stop scripts (Figure 24-49).



**Figure 24-49.** *The start script and scripting methods*

It is interesting to see that one script can invoke the execution of other scripts. This allows you to create complex scripts.

## Clearing

You can clear all your robot trails using the method `clear all pen trails`, which is available in the category `pen use`. You can also clean the traces made by players from the desktop. For that purpose use the last menu item of the menu **appearance**, which you can get from the **World** menu.

## Creating a Tile

If you want to write a script that links two objects, you will need to refer to these objects. In such a case, you will need a *tile* that represents the object to be linked that you can drop into your script. Let us take an example. Imagine that you have two airplanes, one blue and one red, and you want the red airplane to follow the blue one. To do this, you can use the method `move toward`, which is shown in Figure 24-50.



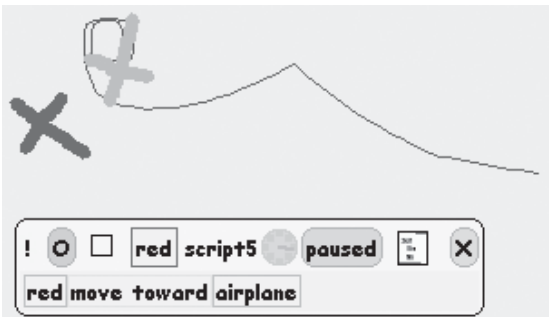
**Figure 24-50.** *The move toward method*

Since you want the red airplane to move toward the blue airplane, and not a dot as in the previous script, first use the orange handle of the blue airplane to get a tile representing it (Figure 24-51).



**Figure 24-51.** *A tile representing the blue airplane*

Then drop this tile into the script. Now the red airplane follows the blue one, as shown in Figure 24-52.



**Figure 24-52.** *The script control panel: the red airplane is following the blue one.*

## Internationalization

If you want to use eToy with small children, you will want everything to appear in their native language. The complete Squeak interface has been translated into a number of languages. Bring up the **World** menu and select the item **help...** followed by **set language...** (Figures 24-53 and 24-54).

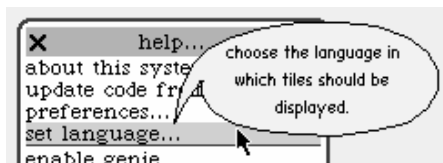


Figure 24-53. The set language menu



Figure 24-54. Choosing Spanish

## Summary

This chapter has presented just a small survey of the many possibilities offered by eToy. I suggest that you go to <http://www.squeakland.org> and check out the material available there.



# A Tour of Alice

**A**lice is an authoring environment for building interactive 3D worlds in Squeak. Alice is also a research project whose goal is to provide abstraction and an environment that is easy for novices to learn and use. Squeak Alice is a port of Alice in Squeak made by Jeff Pierce. A detailed description of the system is given in one of the chapters of the book *Squeak: Open Personal Computing and Multimedia*.<sup>1</sup> Squeak Alice is built on Balloon, the 3D engine of Squeak, which runs on any platform, with no special hardware requirements. This chapter presents Squeak Alice, but within the context of this book.

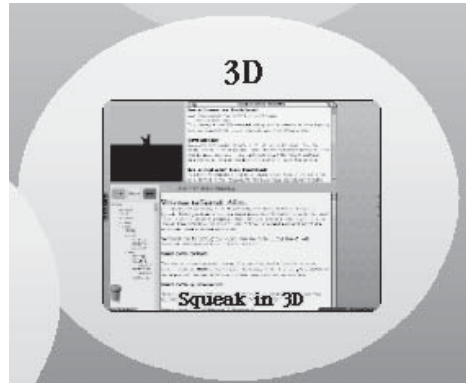
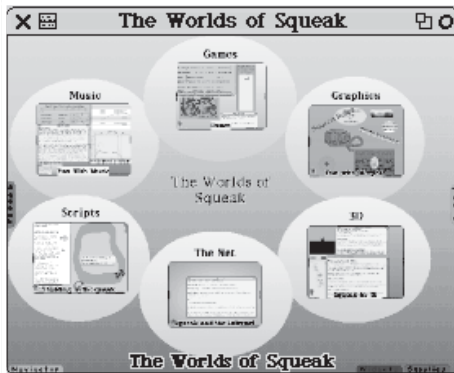
Squeak Alice comes with a complete environment for manipulating 3D objects. To develop scripts and interact with 3D objects, you can either create a new environment, as I will explain later in the chapter, or use the predefined environment that is provided by default in the Squeak environment. To get started faster, I suggest that you use the predefined environment first. Later on, you can open and use your own 3D characters. That is the strategy that we will follow in this chapter.

---

1. Mark J. Guzdial and Kimberly M. Rose, *Squeak: Open Personal Computing and Multimedia* (Prentice Hall, 2001).

## Getting Started with Alice

When you start a standard Squeak environment, there appears a window called *The Worlds of Squeak*, as shown in Figure 25-1. If you click on this window, you arrive at a place containing several small windows, each of them representing a demo related to an aspect of Squeak.



**Figure 25-1.** There are several environments for playing with the multimedia aspects of Squeak.

If you click on the window named 3D, you will arrive in the predefined Squeak Alice environment, as shown in Figure 25-2. There are four windows on the desktop: the *top right-hand* pane is just a window containing some notes indicating where to find the 3D objects and other information. The top left-hand pane, which contains a bunny in 3D, is the 3D world in which 3D objects, called actors, evolve. The bottom right pane is the script editor of Squeak Alice. This is the window that we will use to create 3D objects and define scripts to control these objects. The script editor already contains a long list of interesting scripts that I suggest you try out later. The bottom left-hand pane shows a hierarchical list of all the actors that currently exist in the 3D world.

Before you get started, I suggest that you check the display depth of your environment. The display depth represents the number of colors that you can have. To change it, select **appearance...** in the world menu and then **set display depth...** I recommend that you try the setting 32, but the result may depend on the capacity of your graphics card. When the display depth is not that of your card, Squeak has to transform in “permanence” the rendering of the 3D objects, so Alice gets slower. Once you have set the display depth, I also suggest that you enlarge the left window a bit by bringing up the halo on the complete window and selecting the yellow handle. Note that the complete window is called “camera” when you bring up the halo; that is because this window displays what a camera would be observing in the 3D world.

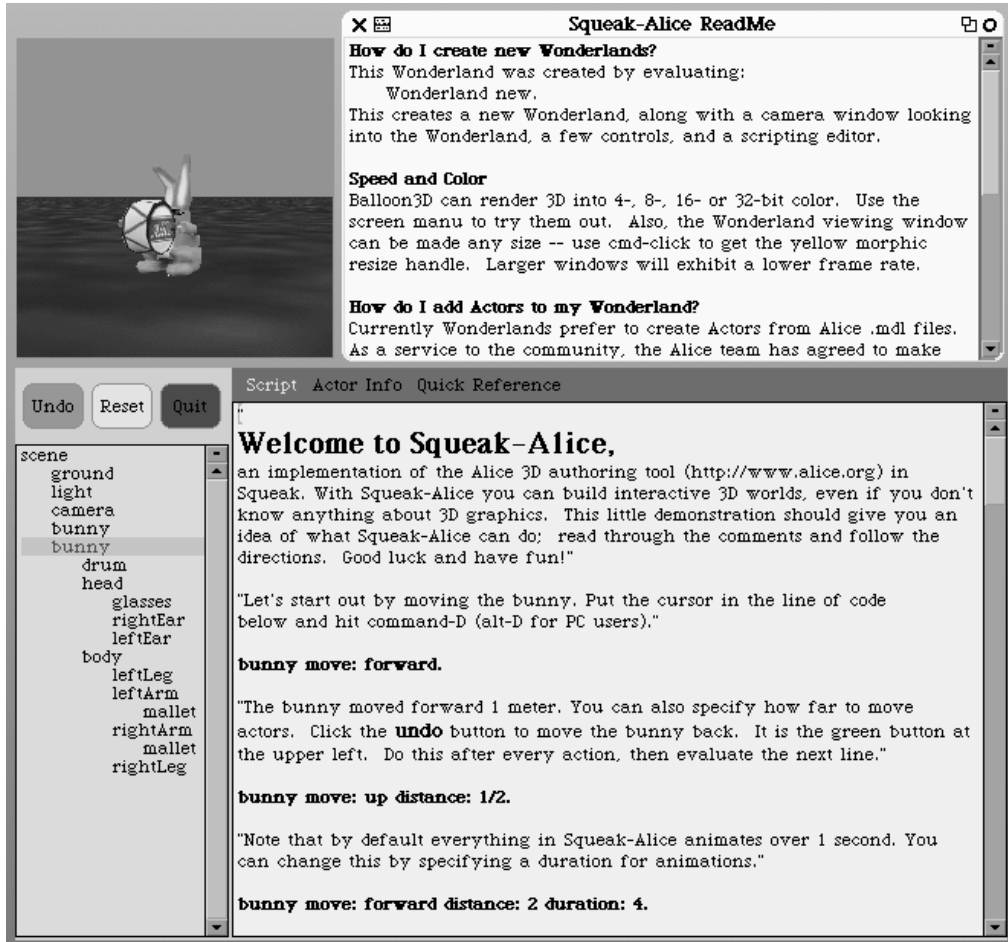


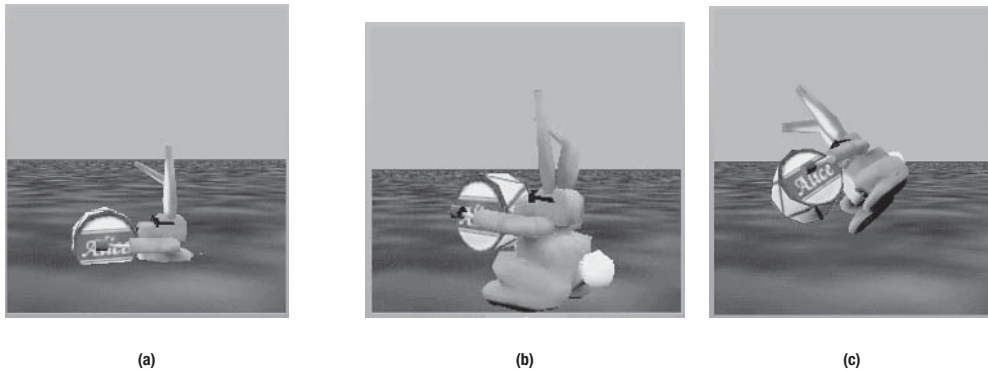
Figure 25-2. The predefined Squeak Alice environment



## Interacting Directly with Actors

With Squeak Alice you can directly interact with the 3D objects, such as the bunny, that exist in the 3D world:

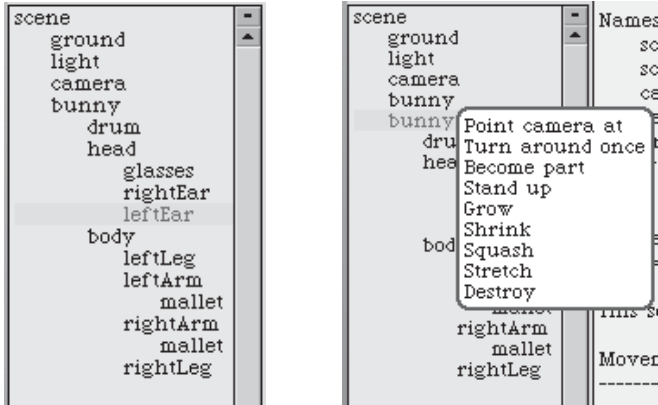
- To move an actor horizontally, click on it in the 3D world. This lets you bring the bunny closer or farther away in the world.
- To move an actor vertically, press Shift+Click (see Figure 25-3 (a)).
- To rotate an actor vertically, press Ctrl+Click (see Figure 25-3 (b)).
- To rotate an actor freely, press Ctrl+Shift+Click (see Figure 25-3 (c)).



**Figure 25-3.** You can move the bunny up and down (*Shift+Click*), rotate it vertically (*Ctrl+Click*), and do a free rotation (*Shift+Ctrl+Click*).

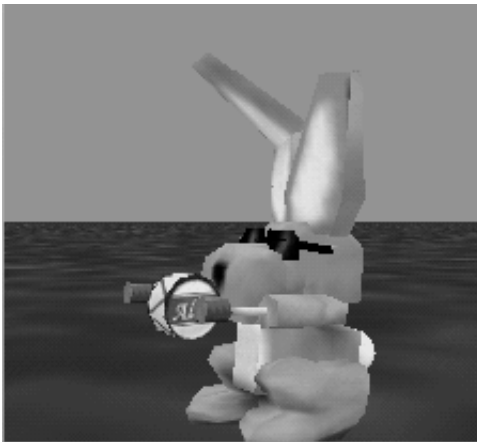
Note that if you want an actor to return to a stable position, use the method `standUp`. This is extremely useful for experimenting with actions performed in parallel.

You can also interact with the actors via the list of all the actors available in the world, as shown in Figures 25-4 and also 25-2. The list contains all the actors. You can see that the light, the camera, and the ground are all actors. You can apply certain predefined actions such as growing, shrinking, and stretching by selecting the actor and obtaining the pop-up menu (see Figure 25-4 (right)).



**Figure 25-4.** A list of all the actors and their hierarchical structure

Actors can be composed of other objects. An actor's parts are hierarchically structured. For example, the bunny is composed of a head and a body. The head is composed of glasses and ears. The parts are displayed in the list as well, and you can apply the actions to them, too. Figure 25-5 shows the bunny after the following transformation: we shrank the drum, enlarged the head, and squished the left ear.



**Figure 25-5.** A transformed bunny

## The Environment

As I have already mentioned, the environment is composed mainly of three graphical components: the scene, or camera window, which displays the 3D world (top-left window in Figure 25-2); the actor list (See Figure 25-4); and the script editor (area at the right of the scene actor list in Figure 25-2). I will now discuss the script editor in detail. There are three buttons at its top: **Script**, **Actor Info**, and **Quick Reference** (see Figure 25-6).

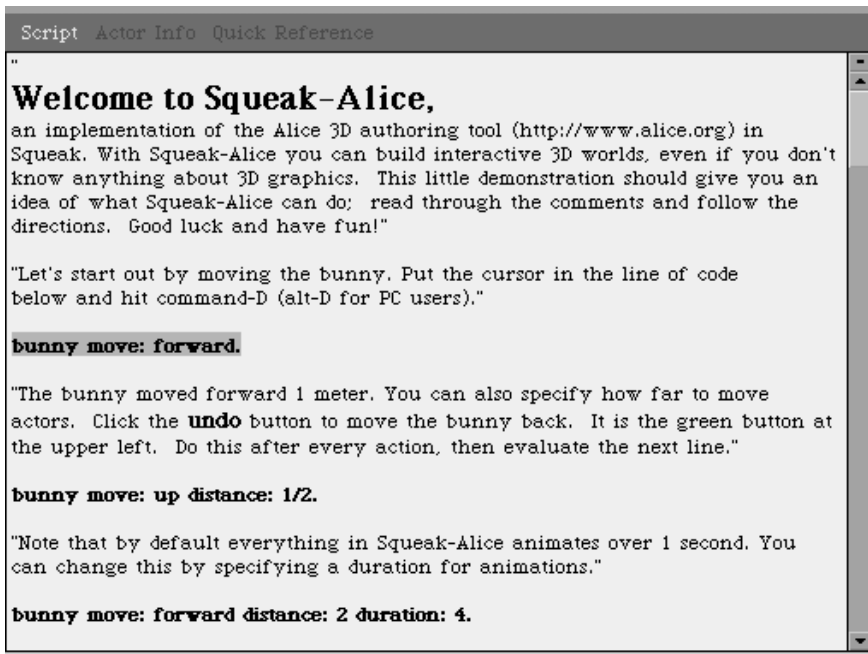
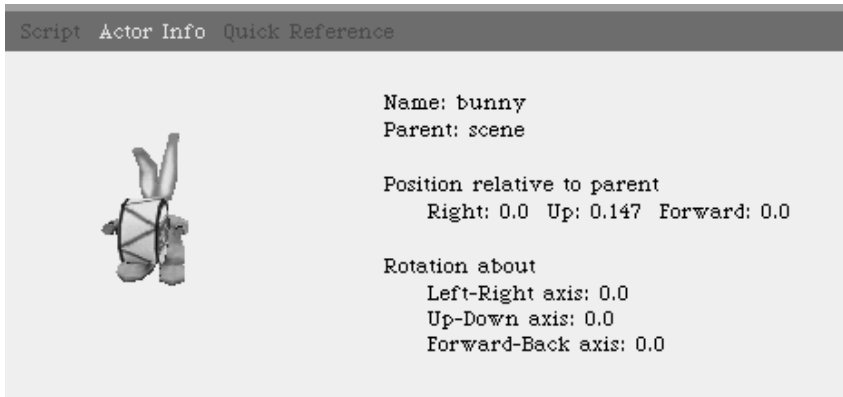


Figure 25-6. The script editor

- The button **Script** allows you to edit and execute scripts.
- The button **Actor Info** shows information related to the actor currently selected in the actor list (Figure 25-7).
- The button **Quick Reference** lists all the possible actions and default constants defined for each kind of action. This is useful online help.

When the **Script** button is selected, you can define scripts and execute them using the traditional **do it** action from the menu or the Command+D or Alt+D shortcut. In fact, the script editor is an extended workspace dedicated to the execution of Alice. This extended workspace contains predefined variables, as explained in the Quick Reference. For our current exploration you need to know only that `camera` refers to the default camera, `cameraWindow` to the scene morph itself, and `w` to the Wonderland, that is, the complete Alice system.

Note that Squeak Alice runs without hardware acceleration, which is disabled by default. As such, Squeak Alice runs on any platform. If you want to turn on hardware acceleration, bring up the menu (red handle) on the camera morph and select the item with the evocative name **hardware acceleration**.



**Figure 25-7.** Actor information

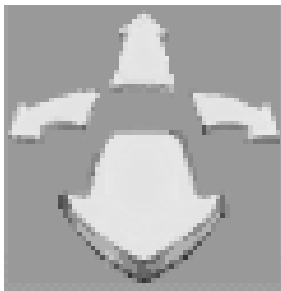
## Scripts

Before proceeding, open the camera controls, as shown in Script 25-1. This way, you will be able to follow the actor if it exists within the vision angle.

### **Script 25-1.** *Adding a camera control*

```
cameraWindow showCameraControls
```

You can move the camera by clicking on the cameraControls widget shown in Figure 25-8. Note that you can move the camera up and down by holding the shift key while moving the mouse over the cameraControls widget. If by accident you press the reset button and you do not want to restart the system, you can load the bunny as explained in the section “Your Own Wonderland” in this chapter.



**Figure 25-8.** Adding a camera control with `cameraWindow showCameraControls`

## Analyzing a First Script

To make it possible to compose actions and change their execution speed, the authors of Alice altered the model of message execution. The model of execution of the actors is different from what we have seen in the rest of the book. Even if the syntax is the same, multiple messages sent to an actor are not executed in sequence but are instead combined. Compare the different types of execution of Script 25-2 by executing it line by line and then by selecting all four lines.

### Script 25-2. *Some simple actions*

```
bunny move: forward.
bunny turn: left.
bunny move: back.
bunny roll: left
```

To execute a script composed of a sequence of actions, use the method `doInOrder:`, as shown in Script 25-3.

### Script 25-3. *Executing a sequence of messages one after the other*

```
w doInOrder: {
 bunny move: forward.
 bunny turn: left.
 bunny move: back.
 bunny roll: left}
```

As you see, the difference between all the effects taking place together and their taking place in sequence is quite important. Another important point is that the Wonderland environment provides some useful predefined constants for programming the actors such as `left`, `back`, and `forward`, which were used in the Script 25-3. Here is a list of some of the available constants for movement, as presented in the Quick Reference pane. I do not present actions related to location in this chapter.

- `direction`: `left`, `right`, `up`, `down`, `forward`, and `back`.
- `duration`: `rightNow` and `eachFrame`.
- `style`: `gently`, `abruptly`, `beginGently`, and `endGently`.
- `position`: `asIs`
- `location`: `onTopOf`, `below`, `beneath`, `inFrontOf`, `inBackOf`, `behind`, `toLeftOf`, `toRightOf`, `onFloorOf`, and `onCeilingOf`.

These constants are used to specify variations of the methods for manipulating actors. Read the Quick Reference to see the possible combinations.

## Moving, Turning, and Rolling

Actors can be manipulated to move, turn, and roll using the methods `move:`, `turn:`, and `roll:`, as shown in Script 25-3. The Quick Reference pane shows that these methods can be further parameterized to yield various results. Here are some examples, but I suggest that you read the chapter on Alice in the book mentioned at the beginning of this chapter and the Quick Reference to learn about all the possibilities. Note that there are some inconsistencies between the description and the implementation, so do not hesitate to experiment.

Script 24-4 presents the list as presented in the Quick Reference.

### Script 25-4. *Variations on move:*

```

move: aDirection
move: aDirection distance: aNumber
move: aDirection distance: aNumber
move: aDirection distance: aNumber duration: aNumber
move: aDirection distance: aNumber duration: aNumber style: aStyle
move: aDirection asSeenBy: anActor
move: aDirection distance: aNumber asSeenBy: anActor
move: aDirection distance: aNumber duration: aNumber asSeenBy: anActor
move: aDirection distance: aNumber duration: aNumber asSeenBy: anActor style: aStyle

move: aDirection speed: aNumber
move: aDirection speed: aNumber for: aNumber
move: aDirection speed: aNumber until: aBlock
move: aDirection speed: aNumber asSeenBy: anActor
move: aDirection speed: aNumber asSeenBy: anActor for: aNumber
move: aDirection speed: aNumber asSeenBy: anActor until: aBlock

```

Here is some explanation: First, you can specify a *distance* using `distance:`. Then you should know that by default, an animation takes one second to execute. To change this default behavior you can specify another *duration* using `duration:` and give the number of seconds that the animation should last. In fact, even if you define a duration of zero, it may not be executed instantaneously by the Wonderland. If you want really instantaneous animations, use the `rightNow` constant. You can also specify a *style*, using `style:` and the associated constants `gently`, which describes how the animation should start or end: `abruptly`, `beginGently`, `endGently`.

Actions are normally time-dependent, which means that they have a start and an end. You can also create persistent actions, using `speed:`, which specifies that actors move at a constant rate. Pay attention that if you use `speed:` but omit a distance, the actor will move forever. The argument of `speed:` for the `move:` method is meters per second, while that for the method `turn:` is the number of turns per second. The argument specified by `for:` allows you to specify a duration for the message when using `speed;` `until:` allows you to specify a condition, expressed by means of a block, during which the action will last. Note that you can stop animation using the message `stop`.

By default, actions such as `move:` and `turn:` take as reference the actor itself. Therefore, when you say `bunny turn: left` the bunny will turn to its left. Sometimes, you will want to specify another frame of reference, and in such a case you should use `asSeenBy:`, which allows you to specify another frame of reference, as shown by the examples in Script 25-5.

**Script 25-5.** *Examples of message variants*

```
bunny move: forward distance: 3 duration: rightNow style: endGently
bunny move: forward distance: 3 duration: 0
bunny move: forward distance: 3 duration: rightNow
bunny move: forward distance: 5 speed: 1
bunny move: left distance: 3 duration: 3 asSeenBy: camera
bunny turn: left turns: 3 speed: 1
bunny roll: right turns: 2
```

## Actor Parts

Actors are composed of parts in a parent–child relationship. Parts belongs to only one parent. This relationship is important because the actions sent to a parent affect its parts. For example, when you ask the bunny to move its head, its glasses, which are part of the head, move too. Parts are nothing special. They are simply actors to which you can send messages, as illustrated in Script 25-6.

**Script 25-6.** *Sending messages to parts*

```
bunny drum roll: left bunny drum roll: left speed: 1

w doInOrder: {
 bunny head glasses move: forward.
 bunny head glasses move: back}

bunny drum stop
```

In fact, all of the actors are parts of a superparent called the *scene*. If you look at the hierarchical list showing all the actors in the World, you see the scene, and below and indented the bunny, but also the ground, the light, and the camera.

Sometimes, you need to be able to send a message to an object affecting only certain of its parts. For example, you might want to be able to change the color of the bunny without changing the color of its left ear; thus you need to make the ear autonomous from the rest of the bunny. To make this possible, Alice introduces the notion of *first class* objects. A first class part belongs to its parent but is not affected when its parent changes. Separating the bunny's head from its torso is illustrated in Figure 25-9.



**Figure 25-9.** *An independent body for cartoon-like animations*

Two methods allow you to control whether an object is part of another one. The method `becomeFirstClass` makes the receiver a first class object, while the method `becomePart` makes the receiver a part of its parent. Execute Script 25-7 line by line to understand the difference.

**Script 25-7.** *First class part examples*

```
bunny setColor: green.
bunny head becomeFirstClass.
bunny setColor: red.
bunny head becomePart.
bunny setColor: pink
```

You can also change the parent–child relationship between objects using the method `becomeChildOf` (See Script 25-8).

**Script 25-8.** *Sending messages to parts*

```
bunny head becomeChildOf: ground
bunny move: forward
ground head turn: left
```

## Other Operations

Actors understand many more messages than what I have shown you so far. Here I give a simple description of the other methods.

### Getting Bigger

The method `resize:` changes the size of the receiver. It exists in multiple variations such as `resize:duration:`, `resizeTopToBottom:leftToRight:frontToBack:`, and `resizelikeRubber:dimension:`, as shown in Script 25-9.



**Script 25-9.** *Resize experiments*

```
bunny resize: 1/2
bunny resizeTopToBottom: 2 leftToRight: 1 frontToBack: 3
bunny resizeLikeRubber: 2 dimension: topToBottom
```

## Quantified Moves

The method `nudge:` moves an actor in multiples of its length, width, or height depending on the direction chosen.

**Script 25-10.** *Nudge experiments*

```
bunny nudge: up distance: 2 duration: 2
```

## Standing Up

The methods `standUp` and `standUpWithDuration: aNumber` allow you to make an actor move to a standing position, which can be useful after experimentation.

## Coloring

The method `setColor:` changes the size of the receiver. It exists in multiple variations such as `setColor:duration:` and `setColor:duration:style:.` Look at Script 25-7 for an example.

## Destruction

The method `destroy` destroys an actor with a nice animation. Fortunately, the Wonderland environment has a powerful undo mechanism.

## Visibility

The methods `hide` and `show` manage the visibility of an object.

## Absolute Moves and Rotations

Up to now we used only actions that change the location or direction of an actor. The method `moveTo:` moves the receiver to a given location, and the method `turnTo:` makes the receiver point in the specified direction. The position and direction may be either a triple in the form `{ right . up . forward }` or an `asIs`. The values of the triple may be a number or `asIs` (e.g., `{ asIs . 0 . asIs }`). Note that the triple describes a location in the actor's parent's reference frame. Hence `bunny moveTo: { 1 . 1 . 0 }` means that the bunny should move 1 meter to the right and 1 meter above the scene's origin, which is the parent of the bunny actor. The same triple in the expression `bunny head moveTo: { 1 . 1 . 0 }` refers to the location 1 meter to the right and 1 meter above the bunny's origin.

Compare the actions of `moveTo:` and `move:` in Script 25-11.

**Script 25-11.** *Absolute move experiments*

```

bunny moveTo: {0 . 0 . 0}
bunny head moveTo: {0 . 1 . 0}.
bunny head moveTo: {0 . -1 . 0}
bunny head move: up.
bunny head move: down
bunny head turnTo: camera duration: 1 style: abruptly
bunny turnTo: camera duration: 1 style: abruptly

```

Note that the method `alignWith: anActor` is equivalent to `turnTo:`.

## Pointing At

The method `pointAt: aTarget` allows you to make actors face each other, where a target may be an actor, a `{ right . up . forward }` trip, or an `x@y` pixel value.

**Script 25-12.** *Experiments with pointAt:*

```

bunny pointAt: camera.
bunny turn: left.
bunny move: forward.
camera pointAt: bunny.

```

## Relative Placement of Actors

Finally, actors can be placed in positions relative to one another using the method `place: aLocation object: anActor`. Locations are specified using the constants `onTopOf`, `below`, `beneath`, `inFrontOf`, `inBackOf`, `behind`, `toLeftOf`, `toRightOf`, `onFloorOf`, and `onCeilingOf`. Try loading multiple actors, as explained later in the chapter. Play with the expressions presented in Script 25-13.

**Script 25-13.** *Placing actors*

```

camera place: inFrontOf object: bunny.
camera move: up.
bunny move: back distance: 2.
camera pointAt: bunny.
bunny head place: toRightOf object: bunny

```

## Time-Related Actions

You can also define actions that are related to the time flow using the `eachFrame` constant as the argument of `duration:`, as shown in Script 25-14. You can use the method `stop` to stop the animation (see the next section).

**Script 25-14.** *Using the eachFrame constant*

```
bunny head pointAt: camera duration: eachFrame.
bunny move: forward
```

You can also specify the length of an action in seconds when the action specifies the `eachFrameFor: aNumber` possibility, as shown in Script 25-15.

**Script 25-15.** *Constraining an action to a certain amount of time*

```
bunny moveTo: {asIs . 0 . asIs} eachFrameFor: 10
```

The method `eachFrameFor: aNumber` makes the actions repeat for the specified number of seconds. The method `eachFrameUntil: aBlock` repeats the actions until the block returns true.

Note that `asIs` is a special constant that states that the method currently executed will not modify the current value. It leaves the value “as is.” However, other methods can change this value. Here the script constrains the bunny to follow along the ground. Note that the triplet means `{Left . Up . Forward}`, and therefore here the bunny cannot move up or down for a period of 10 seconds.

## Animation

To define an animation, just assign it to a variable. For example, I declare in Script 25-16 that `spin` makes the bunny turn twice to the left for a duration of 2 seconds.

**Script 25-16.** *A simple animation*

```
spin := bunny turn: left turns: 2 duration: 2.
```

You can pause an animation (`spin pause`), resume it (`spin resume`), stop it (`spin stop`), or start it again (`spin start`).

Animations can also loop using the methods `loop` and `loop: aLoopNumber`, or be stopped with `stopLooping`.

**Script 25-17.** *A simple animation*

```
flip := bunny turn: forward turns: 1 duration: 2.
```

Now you can compose animations using the method `doInOrder:`, which executes a sequence of messages in sequence, or `doTogether:`, which executes a sequence of messages combined.

**Script 25-18.** *Two simple animations executed in sequence*

```
w doInOrder: {spin start . flip loop:2}
```

**Script 25-19.** *Two simple animations composed together*

```
w doTogether: {spin start . flip loop:2}
```

Note that some combinations do not work, and you may get a walkback (a window telling you that there is a problem). A composition can itself be named and composed with other animations.

**Script 25-20.** *Two simple animations composed in sequence*

```
bla := w doInOrder: {spin start . flip loop:2}. bla start
```

Keep in mind that the method `standUp` makes an actor *stand up* on its feet, which is useful after unexpected results of action combinations.

## Your Own Wonderland

You can create your own Wonderland by executing Script 25-21.

**Script 25-21.** *Opening a new Wonderland*

```
Wonderland new
```

Once you have created your own Wonderland or after pressing the reset button, you should load some 3D objects. The team that develops Alice provides a ZIP archive full of 3D characters at <http://www.cs.cmu.edu/~jpierce/squeak/SqueakObjects.zip> (or look for them at <http://www.apress.com>). Note that these objects are represented in an old format, and therefore trying to load the new Alice objects won't work in Squeak.

As a simple experiment you can also create a simple plane using the expression `w makePlaneNamed: 'myPlane'`. Here is how to load 3D objects on a PC (Script 25-22) and under Mac OS X (Script 25-23).

**Script 25-22.** *Loading new 3D objects on PC*

```
w makeActorFrom: 'Objects\Animals\Bunny.mdl'
```

**Script 25-23.** *Loading new 3D objects under Mac OS X*

```
w makeActorFrom: ':Objects:Animals:PurpleDinosaur.mdl'
```

"if you have some problems on mac use the full path name"

```
w makeActorFrom: 'OSX:Users:ducasse:Alice:Objects:Animals:PurpleDinosaur.mdl'
```

After you load multiple characters such as the snowman and the purple dinosaur (Figure 25-10), you can then write and execute Scripts 25-24 and 25-25.



**Figure 25-10.** *The three amigos* w `makeActorFrom: 'Purpl dinosaur.mdl'`

**Script 25-24.** *Multiple actor script*

```
bunny turn: left
bunny turn: left asSeenBy: snowman
snowman place: inFrontOf object: purpleDinosaur.
```

**Script 25-25.** *A quick glance at the bunny*

```
w doInOrder: {
 purpleDinosaur head pointAt: bunny
 purpleDinosaur head alignWith: purpleDinosaur }
```

## Multiple Cameras and Other Special Effects

Having multiple cameras can slow down the 3D rendering of Alice, but it is worthwhile understanding them for building animation. When a new camera is created, a new view is created as well. A new 3D object representing the new camera is also created. Changing the location of a camera by clicking on it automatically changes the view that displays what the camera sees. In addition, changing the position of the camera using the camera controls widget modifies the location of the camera object.

In Figure 25-11, there are three cameras. I chose to use the right camera to have an overall view of the scene, while the two left cameras are set to provide different close-up views of the bunny. Script 25-26 shows how to add another camera window.

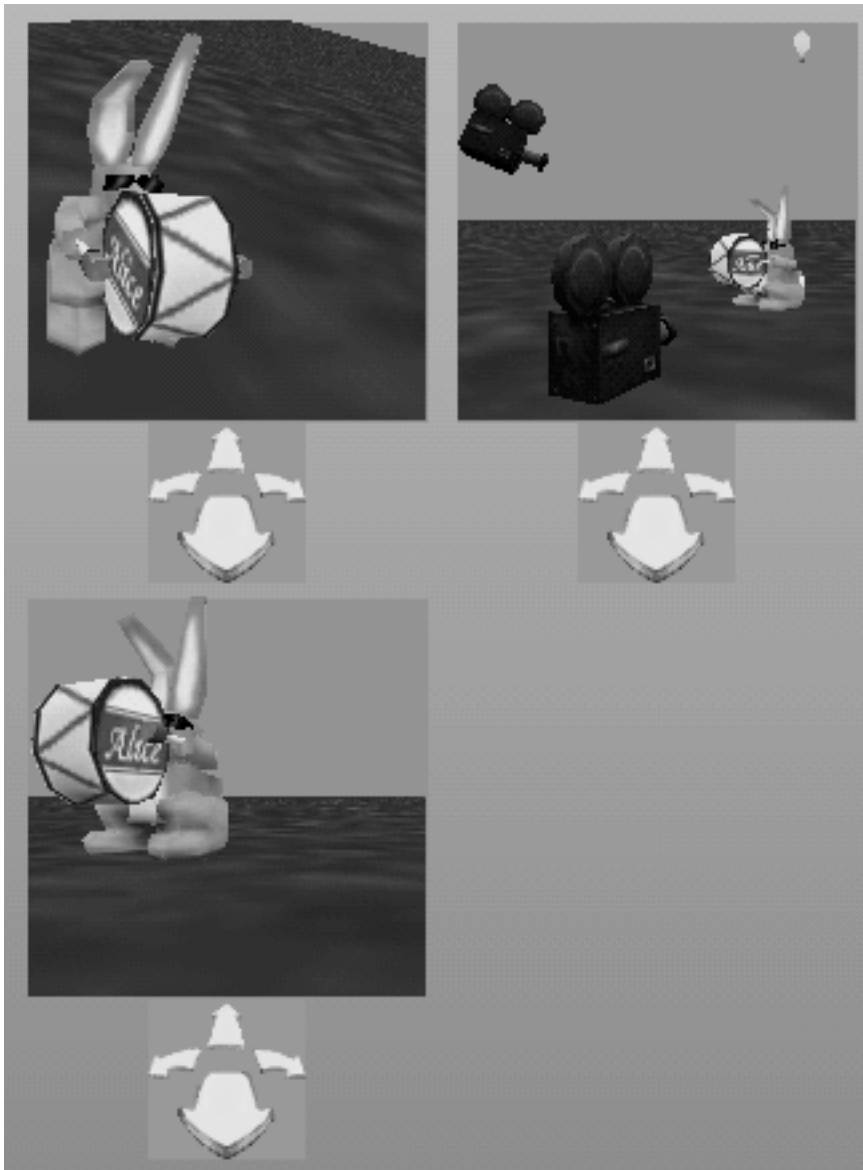
**Script 25-26.** *Creating another camera window*

```
w makeCamera
```

Note that a camera is an actor like any other 3D object, and therefore you can move it using the messages shown in Script 25-27.

**Script 25-27.** *Moving a camera*

```
w doInOrder: {
 camera roll: left.
 camera move: back distance: 4.
 camera standUp.}
```



**Figure 25-11.** *The left windows display what the cameras shown in the right window see.*

## Alarms

Each Wonderland keeps track of the time passing via a scheduler object. When a Wonderland is created, the time is set to zero, and the scheduler starts to update this time at every frame. You can get the current time using the expression `scheduler getTime`. What is interesting is that you can set *alarm* to execute certain actions at a specific point in time using the method `do:at:inScheduler:` or, once a given period of time has passed, using the method `do:in:inScheduler:.` Script 25-28 defines two alarms.

### Script 25-28. *Two alarms*

```
Alarm
do:
 [bunny head turn: left turns: 3.
 bunny setColor: red]
at: (scheduler getTime + 5)
inScheduler: scheduler.
```

```
Alarm
do: [bunny setColor: pink]
in: 8
inScheduler: scheduler.
```

You can send the following messages to an alarm: `checkTime`, which returns the time at which the alarm should be executed, and `stop`, which will stop the alarm if it has not already been executed.

## Introducing User Interaction

At this point you can program animations, but you cannot yet define interactions with the user. Squeak Alice allows you to attach actions to actors when certain events such as mouse clicks occur. For example, Script 25-29 makes the bunny turn its head when it is clicked with the right mouse button.

### Script 25-29. *Defining an action associated with a right mouse click*

```
bunny respondWith: [:event | bunny head turn: left turns: 1] to: rightMouseClicked
```

The three methods `addResponse: aBlock to: eventType`, `removeResponse: aBlock to: eventType`, and `respondWith: aBlock to: eventType` manage the definition of actions. The actions are expressed using blocks and are associated with event types, among which are the following: `keyPress`, `leftMouseDown`, `leftMouseUp`, `leftMouseClicked`, `rightMouseDown`, `rightMouseUp`, and `rightMouseClicked`.

The difference between the methods `addResponse:to:` and `respondWith:to:` is that the first one allows you to define several different actions with the same type of event, while the second one erases the previously defined actions and defines a new one. The method `removeResponse:to:` removes the corresponding actions. An example using `respondWith:to:` appears in Script 25-30.

**Script 25-30.** *Defining an action associated with a left mouse click*

```

bunny
respondWith:
[:event |
bunny head turn: left turns: 2 duration: 2.
w doInOrder: {
 bunny head move: up.
 bunny head move: down}] to: leftMouseButton

```

In Script 25-31 I have added two reactions and then removed the first one so that only the second is executed when you click on the bunny.

**Script 25-31.** *Managing responses*

```

reaction := bunny
addResponse: [:event | bunny head turn: left turns: 1]
to: rightMouseButton.
bunny
addResponse: [:event | bunny head pointTo: { 0. 0. 0}]
to: rightMouseButton.
bunny removeResponse: reaction to: rightMouseButton

```

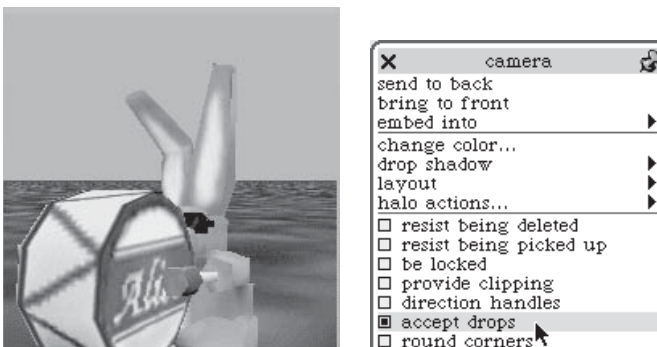
## Hidden Aspects of Alice and Pooh

I would like to finish this short presentation of Squeak Alice by showing you some fun aspects that also illustrate the power of Alice.

### Mapping 2D Morphs to 3D

You can put 2D morphs into 3D objects. The process is as follows:

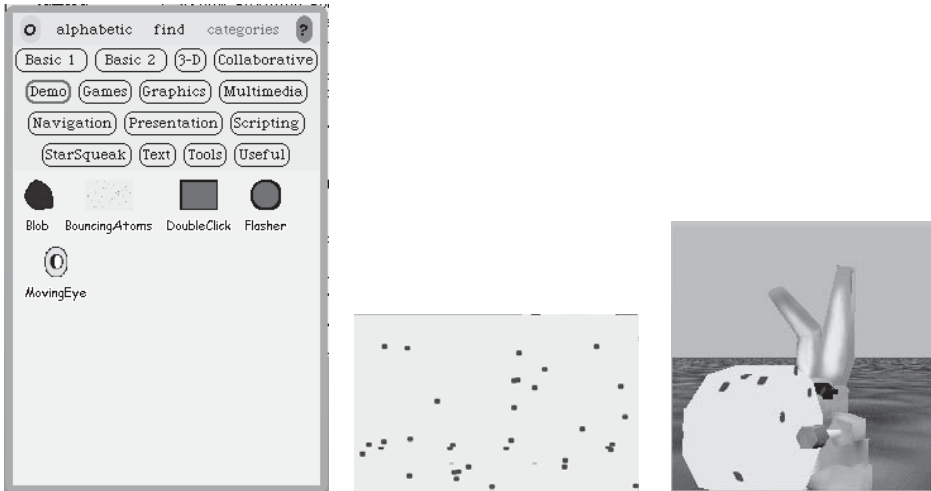
- First bring the red handle **menu** onto the camera window and select the item **accept drops**, as shown in Figure 25-12.



**Figure 25-12.** *Left: The bunny before. Right: Telling the camera to accept other dropped objects*



- Then create a living morph; for example, bring up the object panel using the main Squeak menu item **objects (o)** or Command+O, and create a bouncing atom morph as shown in Figure 25-13.



**Figure 25-13.** *Left: The objects panel. Center: A bouncing atoms morph. Right: A living morph mapped into a 3D object.*

- Drop the newly created bouncing atoms morph onto a part of the bunny. Normally, the morph should be mapped into the 3D object, as shown in the Figure 25-13 (middle).

Finally, Script 25-32 shows a fun and powerful aspect of Alice and Squeak. It creates a wall through which you can see what the mouse is pointing at.

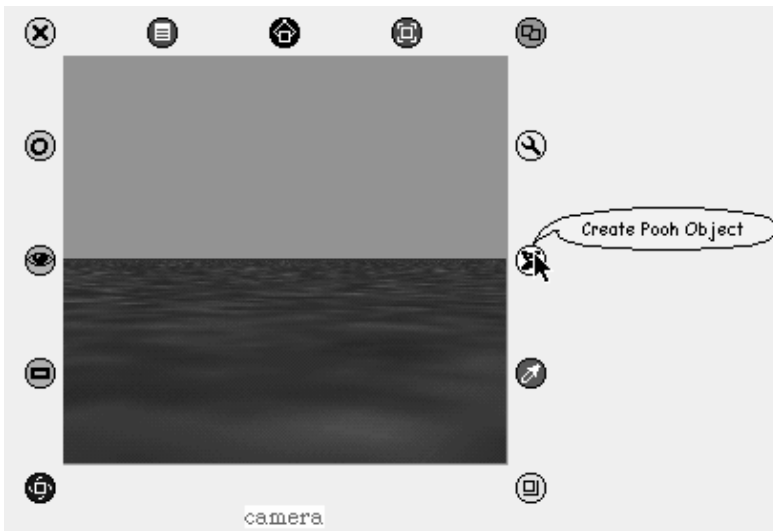
**Script 25-32.** *Fun with Alice*

```
w makePlaneNamed: 'test'.
test
doEachFrame:
 [test setTexturePointer:
 (Form fromDisplay: ((Sensor mousePoint) extent: 50@50)) asTexture]
```

## Pooh: Generating 3D Forms from 2D

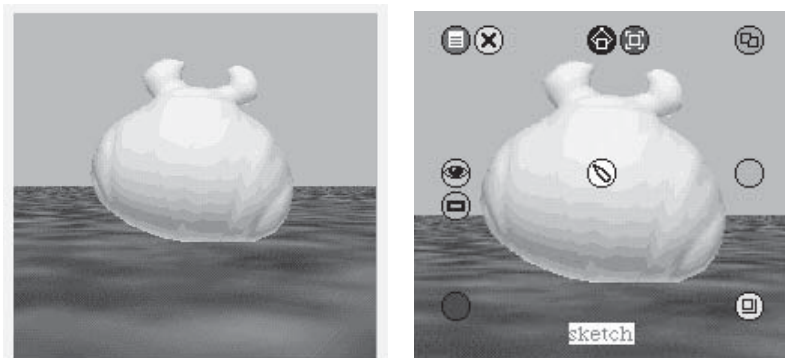
Pooh is a system that allows you to generate 3D forms by drawing 2D forms within an Alice world. If you are curious, try the following steps:

- Open a new Wonderland (Wonderland new).
- Bring up the halo on the camera window, as shown in Figure 25-14.



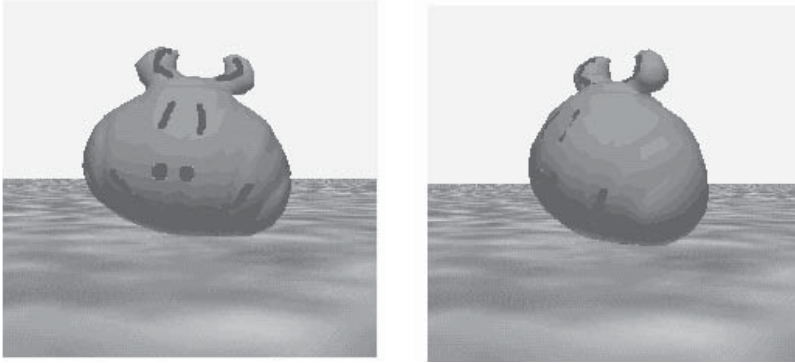
**Figure 25-14.** Opening the halo on the Alice camera to obtain access to Pooh

- Select the middle right white halo with the small bear icon.
- Draw a closed curve directly on the camera window. When you have finished, Pooh generates a 3D form as shown in Figure 25-15 (right).



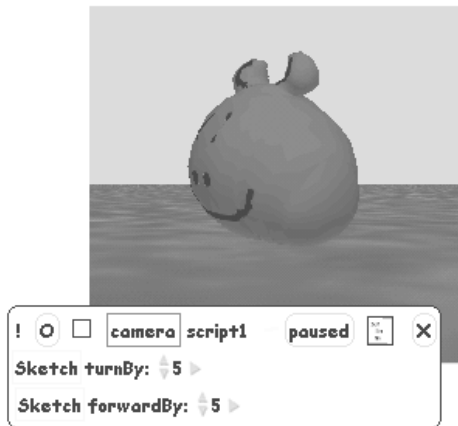
**Figure 25-15.** Left: Obtaining a 3D form. Right: Obtaining the paint editor on a 3D form.

- Now you can paint the form by getting the halo on the new form and selecting the pen halo that appears in the center, as shown in Figure 25-15 (left). This opens a Paint Box. When you are done, press the Keep button of the Paint Box. Figure 25-16 shows the previous shape painted. It also shows that you can rotate the shape, and any other shape.



**Figure 25-16.** *Left: A painted cow. Right: The same cow rotated.*

Finally, I would like to show you some experimental aspects of Squeak. You can use eToy, presented in the previous chapter, with the 3D objects of Alice. You can obtain an eToy viewer using the turquoise handle on the 3D object and use the same techniques as presented in Chapter 24. Figure 25-17 shows a simple script.



**Figure 25-17.** *Using an eToy script to control a 3D object*

## Summary

Alice is a powerful environment. I have shown only a few of the most important aspects. The interested reader should read the chapter dedicated to Alice in the book on Squeak mentioned at the beginning of this chapter.

# Index

## Numbers

- 20 + (2 \* 5) expression, decomposing, 130
- 20 + 2 \* 5 expression, decomposing, 130
- 30-degree angle, turning through, 39
- 45-degree left turn, result of, 42
- (50@60) + (200@400), decomposing, 248
- 50@60 + 200@400, decomposing, 248
- 65 @ 325 extent: 134 @ 100 message, decomposing, 128

## Symbols

- " (double quotes), using in method comments, 142
- ' (single quotes), using with strings, 198
- & (conjunction) message
  - example of, 235–238, 242
  - parentheses error related to, 241
- () (parentheses)
  - including for order of execution, 124–125
  - mistakes caused by, 238–241
  - using with points, 247
- . (period)
  - forgetting, 26
  - using with messages and scripts, 21
- // (division), selecting for steering wheel in eToy, 306–307
- : (colon)
  - using with message selectors, 156
  - using with methods and multiple arguments, 166
  - using with methods and parameters, 160
  - using with parameters, 163
- := (assignment expression)
  - right and left parts of, 103–104
  - using with traces, 204
  - using with variables, 90–91
- @ method, effect of, 248
- [] (square brackets)
  - guidelines for use of, 230
  - using with conditional blocks, 212
  - using with loops, 79, 228
- ^ (caret), returning values with, 144
- | (or) message
  - example of, 235, 237–238, 242
  - parentheses error related to, 241
- || (vertical bars), enclosing variables
  - between, 90

## A

- A
  - drawing, 34–35
  - shape of, 88
  - using absolute moves with, 255–256
  - using variables with, 91–92
  - variations of, 88–89
- Abandon button in debugger window, effect of, 173
- absolute
  - versus relative motion, 249–251
  - versus relative orientation, 40–41, 243
- absolute directions
  - angles associated with, 263
  - significance of, 31–32
- absolute moves
  - making, 248–249
  - significance of, 255–257
- Abstract Art Experiment, 33
- Abstract Design Experiment, 146–147
- abstraction, building over method definition
  - details, 152
- accessor methods, displaying for categories
  - in eToy system, 294
- aCondition method, description and example of, 220
- action length, specifying in Alice, 328
- actions
  - adding for car in eToy, 311
  - composing in Alice, 322
  - creating in Alice, 323
  - defining in Alice, 332
  - removing in Alice, 332–333
- Actor Info button in Alice's script editor,
  - description of, 320
- actor script in Alice, example of, 330
- actors in Alice
  - displaying information about, 321
  - hierarchical structure of, 319
  - interacting with directly, 318–319
  - moving, turning, and rolling, 323–324
  - parts of, 324–325
  - relative placement of, 327
- addResponse:to: and respondWith:to:
  - methods in Alice, comparing, 332
- aDistance method versus goTo: aPoint,
  - 249–251

- airplane
    - adding methods for, 296–297
    - creating joystick for, 297–298
    - dragging and dropping method to create
      - new scripts for, 295
    - drawing, 290
    - experimenting with joystick for, 298
    - flying, 296
    - playing with halo for, 291–295
    - tiling, 313
  - alarms, setting in Wonderland, 332
  - Alice authoring environment. *See also* Wonderland
    - accessing, 315
    - changing parent-child relationship of
      - objects in, 325
    - components of, 320–321
    - composing actions in, 322
    - creating morphs in, 334
    - creating Wonderland in, 329–330
    - defining animations in, 328–329
    - displaying, 317
    - executing messages in, 322
    - features of, 315
    - first class objects in, 324
    - getting time for Wonderland in, 332
    - interacting with actors in, 318–319
    - mapping 2D morphs to 3D in, 333–334
    - message variations in, 324
    - moving, turning, and rolling in, 323–324
    - multiple cameras in, 330–331
    - removing actions in, 332–333
    - resizing receivers in, 325–326
    - script editor in, 320
    - special effects, 330–331
    - specifying action length in, 328
    - stopping animations in, 323
    - time-related actions in, 327–328
    - user interaction in, 332–333
    - using destroy method in, 326
    - using eToy system with, 336
    - using hide and show methods in, 326
    - using moveTo and move methods in,
      - 326–327
    - using nudge: method in, 326
    - using place: method in, 327
    - using pointAt: aTarget method in, 327
    - using scripts in, 321–324
    - using setColor: method in, 326
    - using standUp methods in, 326
  - alternation (or) message, example of, 235, 238, 242
  - angle variable
    - using in methods, 159
    - using with polygons, 97
  - angles
    - for absolute direction messages, 263
    - changing values of, 43
    - linking with wheel position in eToy,
      - 306–307
    - moving through, 42, 43, 44
    - randomizing for animal behavior, 271–272
    - representing, 42
    - versus time, 45
  - angleToPointAt: method
    - defining for animal behavior simulation,
      - 284
    - description and example of, 267
    - diagram of and code for, 266
  - animal behavior simulations
    - decreasing speed in, 280
    - finding food in, 280–286
    - trapped in a box, 274–277
    - of vision, 284–286
    - wandering, 270–274
  - animation elements
    - displaying, 302–303
    - drawing, 300
  - animations
    - creating holders for, 299
    - creating independent bodies for, 325
    - creating sketch recipients for, 301
    - defining in Alice, 328–329
    - stopping in Alice, 323
  - Another Bizarre Staircase Experiment, 113
  - appearance... submenu, description of, 52
  - application file, troubleshooting, 11
  - argument values, viewing with debugger, 177
  - arguments
    - indicating for message receivers, 121
    - in keyword-based messages, 124
    - for methods, 160
    - and parameters, 164–166
    - using with methods, 156
    - variables as, 165
  - Arrows Experiment, 257
  - Art Nouveau Picture Frame Experiment, 147
  - asIs constant, using in Alice, 328
  - assignment, introducing in eToy, 304
  - assignment expression (:=)
    - right and left parts of, 103–104
    - using with traces, 204
    - using with variables, 90–91
  - asString message, example of, 200
- ## B
- bacteria example
    - changing direction in, 277–278
    - increasing speed in, 277
  - balloons, displaying for messages, 7
  - basic button, pressing in eToy, 302–303

- beInvisible and beVisible methods, effects of, 37, 49, 267
  - binary messages. *See also* messages
    - examples of, 119–120, 123
    - explanation of, 119–120
    - lack of priority in, 130
  - black handle for halos, description of, 291
  - blocks, indenting, 81
  - blue handle for halos, description of, 292
  - boldface type, significance of, 170–171
  - Boolean expressions
    - combining, 235–237
    - error in, 239
    - examples of, 234–235
  - Boolean operations, examples of, 238
  - Boolean values, definition of, 234
  - borders, examining in trapped-in-a-box animal behavior, 275–276
  - Bot class
    - associating graphics with, 71–72
    - creating spider robot with, 68–69
    - loading and saving graphics associated with, 70
    - significance of, 23
  - Bot clearImage expression, effect of, 70
  - Bot new east expression, decomposing, 129
  - Bot new expression, executing, 59
  - Bot new go: 100 + 20 message, decomposing, 127
  - Bot new pattern4, execution of, 171
  - Bot Workspace text editor
    - features of, 15
    - obtaining, 52–53
    - saving scripts with, 54
  - BotsInc actions submenu, description of, 52
  - box method, defining star method with, 153
  - boxes. *See also* trapped-in-a-box animal behavior
    - adding exits to, 277
    - moving robots inside of, 264–265
    - variables as, 102–103
  - Boxes Experiment, 153
  - A Broken Square Experiment, 40
  - brown handle for halos, description of, 291
  - bunny
    - moving in Alice, 318
    - sample script of, 330
    - transforming in Alice, 319
  - buttons on mouse, purposes of, 53
- C**
- camera controls, adding and moving in Alice, 321, 330–331
  - capture screen menu item, accessing, 55–56
  - car
    - adding actions for, 311
    - adding sensors to, 308
    - drawing in eToy, 305
    - expressing different behaviors for, 309
    - turning in circle in eToy, 305
  - caret (^), returning values with, 144
  - carriage return characters, representing, 200
  - cascades. *See also* messages
    - sending multiple messages with, 14
    - uses for, 22
  - Case Studies of not-message error, 239
  - categories
    - creating for methods, 139–140
    - displaying in eToy viewer, 294
  - center message, description and example of, 260, 267
  - center versus position, 267
  - centered squares, creating, 184–185. *See also* squares
  - changes file, troubleshooting, 10–11
  - changing directions, 31–33
  - Changing the Reference Direction Experiment, 44
    - characters and strings, 199–200
    - interpreting with tiny language, 219
  - checkerboard construction, creating, 117
  - Checkerboard Squares Experiment, 195
  - circular shape, applying to robots, 64
  - Class Bot Browser
    - features of, 137
    - using, 138
  - classes. *See also* robot factories
    - creating objects for, 30
    - as factories, 30
    - as factories for producing objects, 237–238
    - obtaining new objects from, 23
    - restoring default images to, 70
    - role in object-oriented programming, 17
  - Clear All button, effect of, 54
  - clear all pen trails method, using in eToy, 312
  - Clear Robots button, effect of, 54
  - Clear Trails button, effect of, 54
  - clock, creating, 45
  - Clock Hands Experiment, 43
  - clock hands, moving, 42–43
  - code, indenting, 80–81
  - collection category, looking for, 302–303
  - colon (:)
  - using with message selectors, 156
  - using with methods and multiple arguments, 166
  - using with methods and parameters, 160
  - using with parameters, 163

- color: aColor method, description and example of, 73
  - Color class, effect of, 23
  - Color factory, features of, 64
  - color messages
    - effect of, 8
    - executing, 59
    - names of, 63–64
  - color objects, obtaining, 62
  - color of text, changes in, 27
  - Color r:g:b: expression, creating colors with, 64
  - color sees method, dragging in car example, 310
  - color-based tests, customizing in eToy, 310
  - coloredTurn: anAngle method
    - debugging, 217–218
    - defining, 216
  - colorname color, creating, 35
  - colors
    - asking robots for, 58–59
    - changing for grids, 246
    - changing for robots, 30, 35, 210–212, 214–216
    - creating, 64
    - of pens, 62–63
    - picking with fromUser method, 64
    - relationship to robot direction, 216, 218
  - A Comb Experiment, 85
  - comments, including in methods, 142
  - compass directions, pointing robots in, 32
  - composing methods, definition of, 150. *See also* methods
  - composing solutions, significance of, 183
  - compound Boolean expressions, using, 235–237
  - compound expressions, decomposing with parenthesis priority, 128
  - compound messages
    - definition of, 7
    - examples of, 7–8
    - representing, 125–126
    - sending, 8
  - concentric squares, drawing, 115. *See also* squares
  - conditional block, example of, 212
  - conditional expressions
    - components of, 212
    - nesting, 214–216
    - with one branch, 213–214
    - using to change robot colors, 210–211
    - using traces with, 211–212
  - conditional loops
    - components of, 223–224
    - defining, 226
    - effect of executing once, 226
    - example of, 222–224
  - conditional methods, choosing, 214
  - conjunction (&) message
    - example of, 235–238, 242
    - parentheses error related to, 241
  - constant-angle spiral, definition of, 186
  - A Constant-Angle Spiral Experiment, 186
  - constants for movements in Alice, examples of, 322
  - context-sensitive menus, accessing, 54
  - Controlling the Sides of the Polygon Experiment, 98
  - coordinate system in Smalltalk, description of, 245
  - copying robots, 60
  - copyUpTo method, example of, 200
  - Corridor Experiment, 115, 185
  - Creating and Moving a Robot Experiment, 31
  - cross, drawing with parameters, 158
  - crossWalk1:walk2: method, defining, 162
  - current location of robots, determining, 267
  - cursor value, changing in eToy, 304
  - cut and paste feature, generating regular polygons with, 48
  - cyan handle for halos, description of, 292
- ## D
- daly robot, creating, 30
  - Debug button in debugger window, effect of, 173
  - debugger. *See also* errors; program errors
    - closing, 175
    - definition of, 167
    - fixing parentheses errors with, 240
    - invoking, 172–173
    - printing arguments from, 177
    - using with infinite loops, 226
  - debugger window, opening, 174
  - decomposing problems, significance of, 183
  - Defining Method pattern4 Experiment, 150–151
  - destroy method in Alice, using, 326
  - direction method
    - description and example of, 267
    - example of, 265–266
  - directional convention, significance of, 39
  - directions
    - changing, 31–33, 35, 49
    - obtaining for robots, 262
    - pointing at, 265–266
    - pointing in, 262–263
    - randomizing for animal behavior, 276–277
  - display depth, checking and setting, 316
  - dist: method, obtaining distance with, 280
  - distance
    - comparing in animal behavior simulation, 280–282
    - specifying in Alice, 323



- distanceDetector method
    - changing robot colors with, 210
    - using trace with, 212
  - distanceFrom: aPoint message
    - code for, 263
    - description and example of, 267
  - dividingLine: sideLength method,
    - developing for golden rectangles, 191–192
  - division (//), selecting for steering wheel in eToy, 306–307
  - Do It All button
    - effect of, 15–16, 54
    - executing scripts with, 58
  - Do It button, effect of, 54
  - do it (d) message, effect of, 59
  - doesNotUnderstand: method, effect of, 178
  - double quotes ("), using in method
    - comments, 142
  - doubleFrame method, defining, 151–152
  - doubleFrameWithoutCallingPattern code, 152
  - Doubling the Frame Experiment, 151
  - drawGrids method, using, 246
  - drawing
    - ABC of, 34
    - animation elements, 300
    - capturing, 55–56
    - of equilateral triangle, 46
    - geometric figures, 81–82
    - hexagons, 47
    - of house, 46
    - patterns, 145–147
    - regular polygons, 47
    - robots, 66–67
    - spyglass, 63
    - squares, 136–137, 143
    - staircase, 110–112
    - star, 78–81
  - Drawing a House Experiment, 46
  - Drawing a Rectangle with Arguments Experiment, 161
  - Drawing a Regular Polygon Experiment, 47
  - Drawing a Three-Spoked Figure Experiment, 48
  - Drawing Stylized Crosses Experiment, 162
  - drawings, rotating and zooming, 67
  - duration, specifying for actors in Alice, 323
- E**
- eachFrame constant, using in Alice, 328
  - east message, example of, 32
  - easterly direction, indicating, 42
  - ellipses
    - changing appearance in eToy, 303
    - around messages, meaning of, 120
  - environment
    - components of, 6
    - identifying robots in, 6
    - installing, 4
    - opening, 5–6
    - quitting and saving, 10
  - equilateral triangle, drawing, 46
  - errors. *See also* debugger; program errors
    - fixing, 178–180
    - generating with points, 247–248
    - learning from, 216–218
  - escaping: aBox withExit: aExit method,
    - creating, 277
  - eToy system
    - basic button in, 302–303
    - cars and drivers example in, 305–311
    - changing holder elements in, 303–304
    - clearing robot trails in, 312
    - clearing tiles in, 312–313
    - conditions and tests in, 309–310
    - creating animations in, 299–305
    - creating new scripts with, 295
    - creating road in, 308
    - creating sensors in, 308
    - customizing, 311–314
    - customizing color-based tests in, 310
    - displaying events menu in, 297
    - features of, 289
    - internationalization in, 314
    - linking objects in, 312–313
    - modifying names of sketches in, 293
    - opening viewer in, 292
    - running scripts in, 312
    - steering airplane example in, 290–297
    - using joystick for airplane example in, 297–299
    - using watchers in, 294
    - using with 3D objects in Alice, 336
  - events menu, displaying in eToy system, 297
  - execution stack, example of, 171
  - Experiments. *See also* scripts
    - Abstract Art, 33
    - Another Bizarre Staircase, 113
    - Another Spiral, 186
    - Arrows, 257
    - A Broken Square, 40
    - Changing the Reference Direction, 44
    - Checkerboard Squares, 195
    - A Comb, 85
    - A Constant-Angle Spiral, 186
    - Controlling the Sides of the Polygon, 98
    - A Corridor, 185
    - Creating and Moving a Robot, 31
    - Defining Method pattern4, 150–151
    - Doubling the Frame, 151
    - Drawing a House, 46



Experiments (*continued*)

- Drawing a Rectangle with Arguments, 161
- Drawing a Regular Polygon, 47
- Drawing a Three-Spoked Figure, 48
- Drawing Stylized Crosses, 162
- frAnkenstein, 89
- Golden Rectangle, 94
- Increasing Golden Rectangles, 193
- A Ladder, 85
- A Long Corridor, 115
- A Method for the Art Nouveau Picture Frame, 147
- A Method to Draw a Cross, 158
- A Method to Draw a Hexagon, 158
- Moving Clock Hands, 43
- Mystery Scripts, 38
- PICA, 34
- Placement of the Increment in the Loop, 112
- Putting a Trace inside the Loop, 204
- Pyramid, 117
- A Pyramid with a Variable Number of Terraces, 96
- A Pyramid with a Variable Terrace Size, 96
- A “Real” Clock, 45
- Rectangles 1 and 2, 251
- A Rectangular Pyramid, 194
- A Regular Hexagon, 82
- A Regular Pentagon, 82
- A Relative Square, 40
- Russian Squares, 114
- Russian Squares Experiment, 185
- Scripts That Don’t Work, 95
- A Simple Abstract Design, 146–147
- A Simple Maze, 113
- Some Boxes, 153
- SOS, 31
- A Spiral Out of Lines, 188
- A Spiral with Four Parameters, 187
- “Spirals” Out of Spirals, 188
- Spirals with Constant Distance, 187
- A Square, 33
- Square Ripples in a Square Pond, 184
- A Square Using a Loop, 81
- Squares, 117
- A Staircase, 33, 84
- A Staircase without Risers, 84
- A Staple, 85
- A Star, 153
- A Star with Sixty Branches, 80
- The Step Pyramid of Saqqara, 33
- A Swiss Cross, 84
- A Ten-Step Pyramid, 83
- Tilting the Square, 40
- Translating a Robot by a Point, 259
- Triangle 1, 252
- Triangle 2, 255

- A Triangular Pyramid, 195
- Tumbling Squares, 85
- Using the Methods translate: 1) and translate: 2), 259
- A Variety of A’s, 89
- Your Choice, 153
- expressions
  - components of, 119
  - examples of, 20
  - executing, 60
  - parenthesized equivalents of, 131
  - printing results of, 60
  - relationship to methods, 143
  - relationship to programs, 16, 18
  - self halt expression, 172–173
  - for staircase drawing, 112
  - using square brackets ([]) with, 230
  - using variables in, 105
- extent: aPoint method, description and example of, 73
- extent: widthAndHeight message, resizing robots with, 65

**F**

- factories
  - classes as, 30
  - Color factory, 64
  - relationship to manufactured objects, 17
- false and true objects, returning with Boolean expressions, 234–235
- file lists, loading scripts with, 55
- files
  - importing, 56
  - troubleshooting, 10–11
- findFoodAreaByDistance: aFoodRectangle method, implementing, 280–281
- first class objects in Alice, description of, 324–325
- first method, example of, 199
- flaps
  - definition of, 6
  - obtaining Bot workspaces from, 52
- flying geese, pattern of, 254, 257–258
- folders, navigating, 55
- food, finding in animal behavior simulation, 280–286
- for: method, using in Alice authoring environment, 323
- forward method, dragging and dropping in eToy, 305
- fromUser method, picking colors with, 64
- full-screen mode, going into, 247

**G**

- geometric figures, drawing, 81–82
- getImageFromClass method, description and example of, 73

getter methods, displaying for categories in eToy system, 294

Glossary for methods, 147

go: anInteger ifStayInBox: aRectangle method, code for, 264

go message

- advancing along pixels with, 31
- drawing rectangles with, 14
- effect of, 7

Golden Rectangle Experiment, 94

golden rectangles. *See also* rectangles

- drawing, 189–190
- nesting, 189, 192
- one-line-per-rectangle solution for, 190–193

golden section, calculating, 189

goldenRectangle: methods, examples of

- width method, developing, 191–193

goTo: aPoint message

- versus aDistance, 249–251
- description and example of, 260

graphics

- associating with Bot class, 71–72
- drawing and preserving for robots, 66–67
- loading, 68
- loading and saving, 70
- loading and saving for Bot class, 70
- saving and restoring, 67–72
- using scripts with, 69–72

green handle for halos, description of, 291–292

grey handle for halos, description of, 292

grids, using, 246–247

## H

halo of handles

- accessing, 56, 60
- explanation of, 53–54
- getting information about, 60
- manipulating for airplane, 291–295
- obtaining when drawing robots, 56, 66
- for steering wheel in eToy, 306

handles. *See* halo of handles

heading value, dividing for steering wheel in eToy, 306–307

healthy region, imagining rectangle as, 279

height variable, relationship to width and midheight, 93

heptagon, drawing with polygon100: method, 160

Hexagon Experiment, 82

hexagons

- drawing, 47
- drawing with parameters, 158
- example of, 97

hide and show methods in Alice, using, 326

holder elements, changing in eToy, 303–304

holders

- creating for animations, 299
- dropping pictures in, 301
- opening in viewers, 302

home message, effect of, 264

house, drawing, 46

## I

ifFalse: method, effect of, 213–214

ifStayInBox method, code for, 265

ifTrue:ifFalse: method

- effect of, 211
- using empty conditional block with, 213

image file, troubleshooting, 10–11

images

- applying to robots, 72
- changing, 71
- restoring to classes, 70
- saving, 70

importing files, 56

Increasing Golden Rectangles Experiment, 193

infinite loops, stopping, 226–227

initializing variables, 116

installation

- on Macintosh and Windows systems, 4
- tips for, 4
- troubleshooting, 10–11

interactive application, example of, 228–229

internationalization, implementing in eToy, 314

interpret: aCharacter method, defining, 218

Into button, using with debugger, 176

invisibility and visibility, applying to robots, 35

## J

joystick

- creating for airplane, 297
- experimenting with, 298
- linking with script, 298–299

jump command, using in PICA Experiment, 34

jump: messages

- effect in absolute moves, 248–249
- effect of, 31

jumps, drawing letter A with, 256

jumpTo: aPoint message, description and example of, 260

## K

keepABearing: method, defining, 283

key combinations and mouse button, explanations of, 54

keyword-based messages. *See also* messages

- examples of, 119–120, 123–124
- explanation of, 119–120

**L**

- A Ladder Experiment, 85
- languages, setting in eToy, 314
- left button on mouse, purpose of, 53
- Left to Right order of execution, examples of, 129–131
- left turn by 45 degrees, result of, 42
- length, examining in wandering animal behavior, 272
- letter A
  - drawing, 34–35
  - shape of, 88
  - using absolute moves with, 255–256
  - using variables with, 91–92
  - variations of, 88–89
- Lindemeyer systems, significance of, 218
- line segments, drawing, 31
- lines
  - creating spirals from, 188
  - drawing and coloring, 62–63
  - drawing for star, 78
- loadImage: message
  - description and example of, 73
  - effect of, 69–70
  - versus loadImage: 'fileName', 70
- locations
  - determining, 267
  - jumping to, 249
  - representing as points, 244
  - speculating on, 264–265
- A Long Corridor Experiment, 115
- look\* methods, descriptions and examples of, 73
- lookLike method, creating scripts with, 302
- lookLikeBot message, effect of, 64
- lookLikeCircle message, applying to robots, 64, 69
- lookLikeImage method, effect of, 71
- lookLikeTriangle message, effect of, 64–65
- Loop Increment Experiment, 112
- loops. *See also* nested loops
  - combining variables with, 113–115
  - defining, 227
  - experimenting with, 84–85
  - introducing variables in, 116
  - repeating sequences of messages with, 79–80
  - stopping, 226–227
  - and translations, 257–258
  - using with pyramid script, 83
- lowercase letters, errors related to, 25–26
- M**
- Macintosh
  - installing Squeak on, 4
  - opening environment on, 5
- main menu, options on, 51
- mathematical coordinate system, comparing to Smalltalk, 245
- mathematical message selectors in Smalltalk, priorities of, 131
- Maze Experiment, 113
- mazes, creating, 184–185
- menu items, getting explanations of, 52
- menus, displaying for World, 10
- message execution, looking at, 169–171
- message receivers. *See* receivers
- message selectors
  - for Color class, 63
  - misspelling, 24
  - using colons (:) with, 156
- message sends
  - effect of, 121
  - priority of, 130
- messages. *See also* binary messages; cascades; keyword-based messages; order of execution; unary messages
  - common examples of, 53
  - components of, 119
  - examples of, 7, 20–21
  - executing, 176
  - executing in Alice, 322
  - forgetting periods between, 26
  - identifying, 120–122
  - order of execution of, 240, 247–248
  - results of, 57
  - sending to actor parts in Alice, 324–325
  - sending to robot factories, 9
  - sending to robots, 7–8
  - sending with cascades, 14
  - types of, 119–120
  - using loops with, 79–80
  - using with Boolean expressions, 235–237
  - variations in Alice, 324
- method categories, creating, 139–140
- method definitions, modifying with
  - debugger, 178, 180
- method execution, overview of, 165–166
- A Method for the Art Nouveau Picture Frame Experiment, 147
- A Method to Draw a Cross Experiment, 158
- A Method to Draw a Hexagon Experiment, 158
- methods. *See also* composing methods
  - adding for airplane, 296–297
  - adding traces to, 224–225
  - calling other methods with, 152
  - compiling and testing, 141
  - components of, 142–143
  - debugging, 176
  - defining for golden rectangles, 190–191
  - defining with Class Bot Browser, 138, 140–141
  - defining with multiple arguments, 166

- defining with multiple parameters, 160
  - defining with square method, 153
  - definition of, 22
  - displaying for airplane joystick, 298
  - displaying for categories in eToy system, 294
  - dragging and dropping to create new scripts for airplane, 295
  - for drawing squares, 156–157
  - entering without executing, 176–177
  - recompiling with debugger, 178
  - relationship to expressions, 143
  - versus scripts, 136–137, 143–144
  - using and reusing, 141, 151
  - using arguments with, 156, 160
  - values returned by, 212
  - variables in, 159–160
  - middle button on mouse, purpose of, 53
  - midheight variable, relationship to width and height, 93
  - morphic projects, opening, 289
  - morphs, creating in Alice, 334
  - Morse code, drawing SOS message in, 31
  - motion. *See also* moving robots
    - in Alice authoring environment, 323–324
    - relative versus absolute motion, 249–251
  - mouse buttons
    - explanations of, 53–54
    - and key combinations, 54
  - move: method, using in Alice authoring environment, 323–324
  - move toward method, using in eToy, 312–313
  - movement constants in Alice, examples of, 322
  - moves, making absolute moves, 248–249
  - moveTo and move methods in Alice, using, 326–327
  - Moving Clock Hands Experiment, 43
  - moving robots, 35, 60. *See also* motion
  - Mystery Scripts Experiment, 38
- N**
- n timesRepeat: [], effect of, 79, 86
  - named scripts. *See* methods
  - negated method, using with points, 258
  - negation (not) message, example of, 235–236, 238, 242
  - nested loops, examples of, 194–195. *See also* loops
  - nesting conditional methods, 214–216
  - new message, effect of, 9
  - nil value, significance of, 168–169, 179–180
  - north direction, moving robots in, 223, 228
  - north message, example of, 32
  - northEast message, example of, 32
  - northWest message, example of, 32
  - not (negation) message
    - example of, 235–236, 238, 242
    - parentheses error in, 239
  - nudge: method in Alice, using, 326
  - numberOfSides variable, using in methods, 159
  - numbers and strings, overview of, 201, 203
- O**
- object-oriented programming language, Smalltalk as, 17
  - objects
    - behavior in Smalltalk, 57
    - in binary messages, 123
    - classes as factories for production of, 237–238
    - creating for classes, 30
    - linking in eToy, 312–313
    - manipulating in Alice, 325
    - obtaining from classes, 23
    - relationship to binary messages, 120
  - open... submenu, description of, 52
  - or (!) message
    - example of, 235, 237, 237–238, 242
    - parentheses error related to, 241
  - or (alternation) message
    - example of, 237, 237–238, 242
  - orange handle for halos, description of, 292
  - order of execution. *See also* messages
    - of messages, 240, 247–248
    - overview of, 124–125
    - Rule 1, 124, 125–127, 131
    - Rule 2, 124, 127–128
    - Rule 3, 129–131
  - Over button, using with debugger, 176
- P**
- pablo variable, declaring, 102–103
  - Paint tool, opening to draw airplane, 290
  - painting tool, opening, 66
  - panes of Class Bot Browser, explanations of, 138
  - parallel motion, example of, 249
  - parameter values, changing, 163
  - parameters
    - and arguments, 164–166
    - declaring, 163
    - defining methods with, 160
    - definition of, 156
    - drawing squares with, 163
    - features of, 163
    - naming, 161
    - prohibition of assigning values to, 163
    - using, 158
    - using with polygons, 159
    - using with spirals, 187
    - and variables, 162–163

- parentheses (())
  - including for order of execution, 124–125, 131
  - mistakes caused by, 238–241
  - using with points, 247
- Parentheses First order of execution, examples of, 127–128
- passImageToClass message, effect of, 69
- passImageToClass method, description and example of, 73
- pattern method
  - debugging, 179
  - defining, 169
  - recompiling with debugger, 180
  - selecting in debugger window, 174
  - stepping into, 177
- pattern4 method
  - defining, 150–151, 170
  - selecting in debugger, 175
- patterns
  - drawing, 145–147
  - examples of, 150–151
- pen size and color, overview of, 62–63
- penColor and penSize methods, descriptions and examples of, 73
- penColor: message, effect of, 62–63
- pentagon, drawing with polygon100: method, 160
- pentagon, example of, 96
- Pentagon Experiment, 82
- period (.)
  - forgetting, 26
  - using with messages and scripts, 21
- pica color: Color yellow, decomposing execution of, 125–126
- pica color expression, executing, 58–59
- PICA Experiment, 34
- pica go: 100 + 20 expression, decomposition of, 126
- pica penSize: pica pensize + 2 expression, decomposing, 127
- pica robot
  - applying image to, 72
  - creating, 30
  - as variable, 91
- pica variable, declaring, 102–103, 104–105
- picas, creating for robots, 15, 18–19
- pictures, dropping in holders, 301
- pink handle for halos, description of, 291–292
- pixels
  - determining for forward movement of robot, 105
  - significance of, 19
- place: method in Alice, using, 327
- Placement of the Increment in the Loop Experiment, 112
- plane, creating in Alice, 329
- playerAtCursor method, dragging in eToy, 303
- playfield options menu, displaying in eToy, 311
- point1 \* number message, description and example of, 260
- point1 + point2 message, description and example of, 260
- point1 negated message, description and example of, 260
- pointAt: aPoint method, code for, 265–266
- pointAt: aTarget method in Alice, using, 327
- pointAt: method, using with animal behavior simulation, 283
- pointing and selecting with mouse, 54
- points
  - and absolute moves, 255–257
  - distances from, 263
  - heading toward, 265–266
  - overview of, 244–245
  - for simulating vision in animal behaviors, 280–282
  - translating robots by, 259
  - using negated and setX:setY: methods with, 258
- polygon100: method, using, 160
- polygons
  - automating with variables, 96–97
  - drawing, 47–48
  - with fixed sizes, 98
  - using variables and parameters with, 159
- polygon:size: method, defining, 161
- Pooh system, generating 3D forms from 2D with, 335–336
- PopUpMenu class, example of, 198–199
- position
  - versus center, 267
  - linking with angle in eToy, 306–307
- position message, description and example of, 267
- positionIfGo: aDistance method, code for, 264
- print it (p) menu item, selecting, 59
- printing
  - arguments from debugger, 177
  - results of expressions, 60
- Proceed button, using with debugger, 173, 175, 180
- program errors. *See also* debugger; errors
  - detecting with text colors, 27–28
  - forgetting periods, 26
  - misspelling message selectors, 24
  - misspelling variable names, 24
  - overview of, 23
  - unused variables, 25
  - uppercase versus lowercase, 25–26

- program execution, using Transcript tool
    - with, 202–203
  - programming languages, significance of, 16
  - programs
    - definition of, 16
    - typing and executing, 18
  - Putting a Trace inside the Loop Experiment, 204
  - Pyramid Experiment, 117
  - Pyramid of Saqqara Experiment, 33, 95
    - using loops with, 83
    - using variables with, 96
  - A Pyramid with a Variable Number of Terraces Experiment, 96
  - A Pyramid with a Variable Terrace Size, 96
- Q**
- Quick Reference button in Alice's script editor, description of, 320
  - quitting Squeak environment, 10
- R**
- random direction, choosing for trapped-in-a-box animal behavior, 276–277
  - reactions, managing in Alice, 333
  - read only files, identifying, 11
  - A “Real” Clock Experiment, 45
  - receivers
    - explanation of, 121
    - finding positions in animal behavior simulation, 282
    - hiding and showing, 49
    - indicating arguments for, 121
    - moving if landing inside rectangles, 265
    - resizing in Alice, 325–326
    - returning with square method, 145
    - sending messages to, 144
  - Rectangle Experiments, 251
  - rectangles. *See also* boxes; golden rectangles
    - centering on animals, 274
    - drawing, 14, 264
    - as exits in trapped-in-a-box animal behavior, 277
    - as healthy regions, 279
    - moving robots inside of, 264–265
  - rectangleWidth:height: method, defining, 161
  - A Rectangular Pyramid Experiment, 194
  - red handle for halos, description of, 291
  - red handle painting, item, using with animations, 300
  - reference direction, changing, 44
  - reference lines, drawing, 43
  - regular hexagon, example of, 97
  - A Regular Hexagon Experiment, 82
  - regular pentagon, example of, 96
  - A Regular Pentagon Experiment, 82
  - regular polygons
    - drawing, 47–48
    - with fixed sizes, 98
  - A Relative Square Experiment, 40
  - relative
    - versus absolute motion, 249–251
    - versus absolute orientation, 40–41, 243
  - request:initialAnswer: message, effect of, 199
  - resize: method, using in Alice, 325–326
  - Restart button, using with debugger, 175
  - results of message, significance of, 57
  - right button on mouse, purpose of, 53
  - road
    - coloring in eToy, 310
    - creating in eToy, 308
  - robot clock, creating, 45
  - robot factories. *See also* classes
    - retooling, 68–69
    - sending messages to, 9
  - robot graphics, loading and saving, 70
  - robot movements, tracking in scripts, 203–205
  - robot trails, clearing in eToy, 312
  - robot visibility, controlling, 35
  - robots
    - applying images to, 72
    - changing colors of, 30, 35, 63, 210–212
    - changing directions of, 49
    - changing images of, 71
    - changing shapes and sizes of, 64–65
    - coloring, 214–216
    - copying, 60
    - creating, 9, 22–23, 30, 35
    - creating picas for, 15
    - destroying, 60
    - determining colors of, 58–59
    - determining current locations of, 267
    - determining direction of steps for, 228–229
    - drawing, 66–67
    - identifying in environment, 6
    - interacting with, 8
    - making absolute moves with, 248–249
    - making invisible, 35
    - moving, 35, 60
    - moving inside, 264–265
    - moving north, 223, 228
    - obtaining directions for, 262
    - obtaining information about, 7
    - painting, 66
    - placing in center of screen, 264
    - pointing in compass directions, 32
    - sending messages to, 7–8
    - translating by points, 259
    - turning through given angles, 38
  - rolling in Alice, overview of, 323–324
  - rotating drawings, 67



- rotation, determining for steering wheel in eToy, 306–307
  - Rule 1 of order of execution
    - consequence of, 131
    - explanation of, 124
    - Unary>Binary>Keywords, 125–127
  - Rule 2 of order of execution
    - explanation of, 124
    - Parentheses First, 127–128
  - Rule 3 of order of execution
    - explanation of, 124
    - From Left to Right, 129–131
  - Russian Squares Experiment, 114
- S**
- save contents menu item, accessing, 54
  - save image menu item, accessing, 55–56
  - saveImage: ‘fileName’ message, effect of, 70
  - saveImage method, description and example of, 73
  - saving Squeak environment, 10
  - scheduler getTime expression, using in Wonderland, 332
  - screen regions, capturing, 56
  - screens
    - displaying information on, 198–199
    - placing robots in center of, 264
    - resizing with world menu, 247
  - script editor in Alice, features of, 320
  - scripting method, using in eToy, 312
  - scripts. *See also* Experiments
    - adding traces to, 204
    - analyzing, 18–19, 104–107
    - analyzing in Alice, 322
    - creating for airplane, 295
    - creating with lookLike method, 302
    - declaring variables for, 35
    - definition of, 13
    - executing, 58–60
    - for golden rectangle, 94
    - linking airplane joystick with, 298–299
    - loading, 55
    - versus methods, 136–137, 143–144
    - relationship to expressions, 18
    - running in eToy, 312
    - saving with Bot workspace, 54
    - selecting complete text of, 58
    - using in Alice authoring environment, 321–324
    - using with graphics operations, 69–72
    - writing, 15–16
  - Scripts That Don’t Work Experiment, 95
  - selecting with mouse, 54
  - self halt expression, opening debugger with, 172–173
  - self variable, relationship to methods, 144, 166
  - sensor for car, creating in eToy, 308
  - set language menu, displaying in eToy, 314
  - setColor: method in Alice, using, 326
  - setter methods, displaying for categories in eToy system, 294
  - setX:setY method, using with points, 258
  - shapes of robots, changing, 64–65
  - sideLength parameter
    - declaring, 163
    - using in Russian Squares Experiment, 114
    - using with square method, 157
  - sides of turns, examining in wandering animal behavior, 273
  - sides variable, using with polygons, 97
  - A Simple Abstract Design Experiment, 146–147
  - A Simple Maze Experiment, 113
  - simulating animal behavior. *See* animal behavior simulations
  - single quotes (’), using with strings, 198
  - size method, example of, 199
  - size variable, declaring, 168
  - sizes of robots, changing, 64–65
  - sizeValue parameter, example of, 161
  - sketch names, modifying in eToy, 293
  - sketch recipients, creating for animations in eToy, 301
  - Smalltalk programming language, 30
    - behavior of objects in, 57
    - blocks in, 79
    - coordinate system in, 245
    - laying out code in, 80–81
    - mathematical message selectors in, 131
    - naming variables in, 102
    - significance of, 16–17
  - SOS message, drawing in Morse code, 31
  - sources file, troubleshooting, 11
  - south message, example of, 32
  - southEast message, example of, 32
  - southWest message, example of, 32
  - space characters, representing, 200
  - special effects, using in Alice authoring environment, 330–331
  - speed: method, using with actions in Alice, 323
  - spider robot, creating with Bot class, 68–69
  - spider.frm image file, contents of, 69–70
  - spiral, drawing, 114
  - A Spiral Out of Lines Experiment, 188
  - A Spiral with Four Parameters Experiment, 187
  - spirals, creating, 186
  - “Spirals” Out of Spirals Experiment, 188
  - Spirals with Constant Distance Experiment, 187

- spyglass, drawing, 63
  - square brackets ([])
    - guidelines for use of, 230
    - using with conditional blocks, 212
    - using with loops, 79, 228
  - A Square Experiment, 33
  - square: method
    - defining methods with, 153
    - execution of, 166
    - reproducing pictures with, 194–195
    - returning message receiver with, 145
    - using `sideLength` parameter with, 157
  - square script, using variable with, 162
  - A Square Using a Loop Experiment, 81
  - squares. *See also* centered squares; concentric squares
    - drawing, 40, 81, 136–137, 143
    - drawing with methods, 156–157
  - Squares Experiment, 114, 117
  - Squeak
    - downloading, 4
    - files required for, 10–11
    - installation, 4–5
    - interacting with, 53–54
    - quitting and saving environment of, 10
    - and Smalltalk programming language, 16–17
  - Squeak application file, troubleshooting, 11
  - stack in debugger, stepping through, 175–178
  - staircase
    - drawing, 110–112
    - drawing with treads, 203–205
    - interactive version of, 229
  - Staircase Experiments, 33, 84, 113
  - standUp method, applying to actors in Alice, 318, 326
  - A Staple Experiment, 85
  - star, drawing, 78–81
  - A Star Experiment, 153
  - A Star with Sixty Branches Experiment, 80
  - start script method, using in eToy, 312
  - steering wheel
    - drawing in eToy, 305
    - using heading of, 306–307
  - The Step Pyramid of Saqqara Experiment, 33
  - steps
    - determining directions of, 228–229
    - drawing for staircase, 111–112
  - strings
    - and characters, 199–200
    - and numbers, 201, 203
    - overview of, 198
  - submenus, explanations of, 52
  - A Swiss Cross Experiment, 84
- T**
- tab characters, representing, 200
  - A Ten-Step Pyramid Experiment, 83
  - terraceNumber variable, using with pyramid, 96
  - terraceSize variable, using with pyramid, 96
  - text color, changes in, 27
  - text editor, using Bot Workspace as, 15
  - three-spoked figure, drawing, 48
  - tiles, creating in eToy, 312
  - tiling squares, 194–195
  - tiltedPattern method, significance of, 151
  - Tilting the Square Experiment, 40
  - time
    - versus angles, 45
    - getting for Wonderland in Alice, 332
  - time-related actions, defining in Alice, 327–328
  - timesRepeat: method
    - argument of, 80
    - displaying in debugger, 175
    - effect of, 79, 86
  - tiny languages, interpreting, 218–219
  - totalLength variable, using with polygons, 98
  - traces
    - adding to methods, 224–225
    - drawing for airplane, 296
    - estimating for wandering animal behavior, 272
    - generating, 203–205
    - for simulating vision in animal behaviors, 284–285
    - using with conditional expressions, 211–212
  - Transcript tool
    - generating traces of programs with, 203–205
    - using, 202–203
    - using with conditional expressions, 211–212
    - using with upTo100 method, 225
  - Translating a Robot by a Point Experiment, 259
  - translations
    - of flying geese, 254–255
    - and loops, 257–258
    - overview of, 252–253
    - of triangles, 253–254
  - trapped-in-a-box animal behavior. *See also* boxes
    - flying to opposite border for, 276
    - following borders for, 275–276
    - introducing exit in box for, 277
    - overview of, 274–275
    - random direction for, 276



treads  
 drawing for staircase, 203–205  
 measuring for staircase drawing, 111–112  
 Triangle 1 Experiment, 252  
 Triangle 2 Experiment, 255  
 triangles  
 drawing, 46  
 translating, 253–254  
 A Triangular Pyramid Experiment, 195  
 triangular shape, applying to robots, 64  
 troubleshooting installation of Squeak, 10–11  
 true and false objects, returning with  
 Boolean expressions, 234–235  
 Tumbling Squares Experiment, 85  
 turn: method  
 adding for airplane, 296  
 dragging and dropping in eToy, 305  
 effect of, 39  
 using in Alice authoring environment,  
 323–324  
 turning in Alice, overview of, 323–324  
 turnLeft message, effect of, 7, 38–39  
 turnRight: method, example of, 38–39, 42–43  
 turns, examining in animal behavior,  
 273–274, 275–276  
 turnTo: aDirection message, effect of,  
 262–263, 267  
 turnTo: anAbsoluteAngleInDegrees method,  
 code for, 263

## U

unary messages. *See also* messages  
 examples of, 121, 122  
 explanation of, 119–120  
 order of execution of, 125  
 Unary>Binary>Keywords order of execution,  
 examples of, 125–127  
 underlined message receivers, meaning of,  
 120–121  
 undrawGrids method, using, 246  
 until: method, using in Alice authoring  
 environment, 323  
 uppercase letters, errors related to, 25–26  
 upTo100 method  
 adding trace to, 224–225  
 effect of, 222  
 upTo100Infinite method, executing, 227  
 user input loop, adding to interactive  
 staircase, 229  
 user interaction, implementing in Alice,  
 332–333  
 users, communicating with, 198–199

## V

v method, invoking, 222  
 values  
 assigning to variables, 90  
 changing for variables, 116  
 excluding from variables, 106  
 modifying for variables using eToy, 294  
 relationship to variables, 90  
 representing as nil, 168–169  
 returning, 144–145  
 returning from methods, 212  
 variable declaration, relationship to  
 messages, 21  
 variable names  
 misspelling, 24  
 self variable, 144  
 significance of, 103  
 variable values, changing, 163  
 variables  
 as arguments, 165  
 assigning values to, 90  
 automating polygons with, 96–97  
 as boxes, 102–103  
 changing values of, 116  
 combining with loops, 113–115  
 declaring, 90  
 declaring, initializing, and using, 104–105  
 declaring and assigning, 99  
 declaring for scripts, 35  
 default values of, 168–169  
 defining with walkLength variable, 107  
 definition of, 90  
 errors related to, 24–25  
 experimenting with, 94–96  
 expressing relationships between, 93–94  
 initializing, 90–91, 116  
 introducing in loops, 116  
 for length of pica's walk, 105  
 in methods, 159–160  
 modifying values of using eToy, 294  
 naming, 102  
 and parameters, 162–163  
 as placeholders, 102, 104  
 power of, 92–93  
 referring to, 91  
 using, 116  
 using in expressions, 105  
 using with letter A, 91–92  
 using with Pyramid of Saqqara  
 Experiment, 96  
 using with square script, 162  
 using with staircase drawing, 111  
 using without values, 106  
 A Variety of A's Experiment, 89

vertical bars (| |), enclosing variables  
 between, 90

viewer

- displaying categories in, 294
- opening holder in, 302
- opening in eToy system, 292
- parts of, 293
- on sketch recipient, 301

visibility and invisibility, applying to robots,  
 35

vision, simulating for animal behavior,  
 284–286

VM (virtual machine), Squeak application  
 file as, 11

vocabulary pane, listing common messages  
 in, 53

**W**

w makePlaneNamed: expression, using in  
 Alice, 329

walkLength variable

- changing twice, 106
- declaring, 104–105
- defining variable with, 107
- initializing, 107–108
- writing and reading, 104

wandering behavior, simulating, 270–274

watchers, using in eToy, 294

websites

- eToy system, 289
- Squeak, 4

west message, example of, 32

wheel. *See* steering wheel

whileFalse: loop

- components of, 223
- description and example of, 231
- effect of, 222

whileTrue: loop

- components of, 223
- converting to whileFalse:, 225
- description and example of, 231
- effect of, 222

widgets flap, opening to draw airplane, 290

width variable, relationship to midheight and  
 height, 93

Windows

- installing Squeak on, 4
- opening environment on, 5

Wonderland. *See also* Alice authoring  
 environment

- alarms in, 332
- creating in Alice, 329–330
- opening for use with Pooh, 335

words, selecting in scripts, 58

World menu

- displaying menu for, 10
- opening morphic projects from, 289
- resizing screens with, 247

The Worlds of Squeak window, displaying,  
 315

worm, animating, 300

**X**

x @ y message, description and example of,  
 260

**Y**

yellow handle for halos, description of, 292

yertle robot, appearance of, 72

Your Choice Experiment, 153

**Z**

zooming drawings, 67