Table of Contents

Part 1 Introduction to Interactive Program Design 1

Chapter 1 Introduction to Program Design 3
- 1.1 Computers and Programs 4
- 1.2 Thinking Like a Programmer 6
- 1.3 Programming Primitives, Briefly 8
- 1.4 Ongoing Computational Activity 9
- 1.5 Coordinating a Computational Community 11
  - 1.5.1 What Is the Desired Behavior of the Program? 12
  - 1.5.2 Who Are the Entities Who Interact to Produce the Program's Desired Behavior? 13
  - 1.5.3 What Goes Inside Each Entity (How Does It Work)? 13
  - 1.5.4 How Do These Entities Interact? 14
- 1.6 The Development Cycle 16
- 1.7 The Interactive Control Loop 18
- Chapter Summary 19
- Exercises 20

Chapter 2 The Programming Process 23

Interlude A A Community of Interacting Entities 69
- A.1 Introduction: Word Games 70
- A.2 Designing a Community 71
  - A.2.1 A Uniform Community of Transformers 71
  - A.2.2 The User and the System 73
  - A.2.3 What Goes Inside 75
- A.3 Building a Transformer 76
  - A.3.1 Transformer Examples 77
  - A.3.2 Strings 78
    - A.3.2.1 String Concatenation 79
    - A.3.2.2 String Methods 79
  - A.3.3 Rules and Methods 82
  - A.3.4 Classes and Instances 83
  - A.3.5 Fields and Customized Parts 85
  - A.3.6 Generality of the approach 87
- Chapter Summary 89
- Exercises 90

Part 2 Entities and Interactions 91

Chapter 3 Things, Types, and Names 93
- 3.1 Things in Programs 94
- 3.2 Most Java Things are Objects 95

Table of Contents

Chapter 3 Things, Types, and Names
3.2.1 Doing Things with Objects....96
3.3 Naming Things....99
3.4 Types....103
3.4.1 What a Type Is....103
3.4.1.1 Two Kinds of Types: Primitive Types and Object Types....104
3.4.1.2 Object Types....104
3.4.1.3 What a Type Is: Summary....105
3.4.2 Types of Objects....106
3.4.3 Types of Names....106
3.4.3.1 Declarations and the Type–of–Thing Name–of–Thing Rule....106
3.4.3.2 Definition = Declaration + Assignment....107
3.5 Names for Objects: Label Names....108
3.6 Primitive Types, Literals and Dial Names....110
3.6.1 Literals....111
3.6.2 Primitive Types....112
3.6.3 Names for Primitive–Type Things: Dial Names....115
3.7 A Tale of Things and Names....120
Chapter Summary....124
Exercises....125

Chapter 4 Specifying Behavior: Interfaces....131
4.1 Interfaces are Contracts....132
4.1.1 Generalized Interfaces and Java Interfaces....133
4.1.2 A Java Interface Example....134
4.2 Method Signatures....135
4.2.1 Name....136
4.2.2 Parameters and Parameter Types....136
4.2.3 Return Type....137
4.2.4 Putting It All Together: Abstract Method Declaration Syntax....137
4.2.5 What a Signature Doesn't Say....138
4.3 Interface Declaration....140
4.3.1 Syntax....140
4.3.2 Method Footprints and Unique Names....141
4.3.3 Interfaces are Types: Behavior Promises....143
4.3.4 Interfaces are Not Implementations....144
Chapter Summary....146
Exercises....147

Chapter 5 Expressions: Doing Things with Things....149
5.1 Simple Expressions....150
5.1.1 Literals....150
5.1.2 Names....151

Table of Contents

Chapter 5 Expressions: Doing Things with Things
5.2 Method Invocation....152
5.3 Combining Expressions....153
5.4 Assignments and Side−Effecting Expressions....155
5.5 Other Expressions That Use Objects....156
5.5.1 Fields....156
5.5.2 Instance Creation....157
5.5.3 Type Membership....158
5.6 Complex Expressions on Primitive Types: Operations....158
5.6.1 Arithmetic Operation Expressions....160
5.6.2 Explicit Cast Expressions....162
5.6.3 Comparator Expressions....164
5.6.4 Logical Operator Expressions....165
5.7 Parenthetical Expressions and Precedence....166
Chapter Summary....171
Exercises....172

Chapter 6 Statements and Rules....175
6.1 Statements and Instruction−Followers....176
6.2 Simple Statements....177
6.3 Declarations and Definitions....178
6.4 Sequence Statements....180
6.5 Flow of Control....183
6.5.1 Simple Conditionals....183
6.5.2 Simple Loops....187
6.6 Statements and Rules....189
6.6.1 Method Invocation Execution Sequence....190
6.6.2 Return....191
Chapter Summary....194
Exercises....195

Interlude B Expressions and Statements....197
B.1 The Problem....198
B.2 Representation....200
B.3 Interacting with the Rules....202
B.4 Paying Attention to the World....204
B.5 Fancy Dot Tricks....207
B.6 Remembering State....208
B.6.1 Fields....209
B.6.2 Fields versus Local Variables....211
Chapter Summary....213
Exercises....214

Table of Contents

Chapter 7 Building New Things: Classes and Objects......215
7.1 Classes are Object Factories......216
7.1.1 Classes and Instances......216
7.1.2 Recipes Don't Taste Good......217
7.1.3 Classes are Types......218
7.2 Class Declaration......219
7.2.1 Classes and Interfaces......220
7.3 Data Members, or Fields......221
7.3.1 Fields are Not Variables......223
7.3.1.1 Hotel Rooms and Storage Rental......224
7.3.1.2 Whose Data Member Is It?......224
7.3.1.3 Scoping of Fields......225
7.3.2 Static Members......227
7.4 Methods......228
7.4.1 Method Declaration......229
7.4.2 Method Body and Behavior......230
7.4.3 A Method ALWAYS Belongs to an Object......231
7.4.4 Method Overloading......232
7.5 Constructors......235
7.5.1 Constructors are Not Methods......235
7.5.2 Syntax......236
7.5.3 Execution Sequence......238
7.5.4 Multiple Constructors and the Implicit No−Argument Constructor......239
7.5.5 Constructor Functions......240
Chapter Summary......243
Exercises......244

Part 3 Refining Designs......247

Chapter 8 Designing with Objects......249
8.1 Object Oriented Design......250
8.1.1 Objects are Nouns......250
8.1.2 Methods are Verbs......251
8.1.3 Interfaces are Adjectives......251
8.1.4 Classes are Object Factories......252
8.1.5 Some Counter Code......254
8.1.6 Public and Private......255
8.2 Kinds of Objects......257
8.2.1 Data Repositories......257
8.2.2 Resource Libraries......262
8.2.3 Traditional Objects......264
8.3 Types and Objects......266
8.3.1 Declared Type and Actual Type......266

Table of Contents

Chapter 8 Designing with Objects
8.3.2 Use Interface Types....267
8.3.3 Use Contained Objects to Implement Behavior....268
8.3.4 The Power of Interfaces....269
Chapter Summary....271
Exercises....272

Chapter 9 Animate Objects....273
9.1 Animate Objects....274
9.2 Animacies are Execution Sequences....275
9.3 Being Animate–able....276
9.3.1 Implementing Animate....276
9.3.2 AnimatorThread....278
9.3.3 Creating the AnimatorThread in the Constructor....279
9.3.4 A Generic Animate Object....281
9.4 More Details....282
9.4.1 AnimatorThread Details....282
9.4.2 Delayed Start and the init Trick....285
9.4.3 Threads and Runnables....287
9.4.4 Thread Methods....287
9.5 Where Do Threads Come From?....288
9.5.1 Starting a Program....289
9.5.2 Why Constructors Need to Return....292
Chapter Summary....294
Exercises....295

Chapter 10 Inheritance....297
10.1 Derived Factories....298
10.1.1 Simple Inheritance....299
10.1.2 The java.lang.Object Type....300
10.1.3 Superclass Membership....302
10.2 Overriding....304
10.2.1 The super Expression....305
10.2.2 The Outside–In Rule....306
10.2.3 Problems with Private....307
10.3 Constructors are Recipes....308
10.3.1 The this() Expression....308
10.3.2 The super() Expression....309
10.3.3 Implicit super()....310
10.3.4 Multiple Views....312
10.4 Interface Inheritance....312
10.5 Relationships Between Types....313
Chapter Summary....316

Table of Contents

Chapter 10 Inheritance
Exercises....317

Chapter 11 When Things Go Wrong: Exceptions....319
11.1 Exceptional Events....320
11.1.1 When Things Go Wrong....320
11.1.2 Expecting the Unexpected....321
11.1.3 What's Important to Record....323
11.2 Throwing an Exception....324
11.3 Catching an Exception....329
11.4 Throw versus Return....332
11.5 Designing Good Test Cases....334
Chapter Summary....336
Exercises....337

Part 4 Refining Interactions....339

Chapter 12 Dealing with Difference: Dispatch....341
12.1 Conditional Behavior....342
12.2 Keywords if and else....344
12.2.1 Basic Form....344
12.2.2 The else Keyword....346
12.2.3 Cascaded if Statements....349
12.2.4 Many Alternatives....351
12.3 Limited Options: switch....354
12.3.1 Constant Values....354
12.3.1.1 Symbolic Constants....355
12.3.1.2 Using Constants....356
12.3.2 Syntax....359
12.3.2.1 Basic Form....359
12.3.2.2 The Default Case....363
12.3.2.3 Variations....364
12.3.2.4 Switch Statement Pros and Cons....365
12.4 Arrays....367
12.4.1 What is an Array?....367
12.4.2 Manipulating Arrays....371
12.4.2.1 Stepping Through an Array Using a for Statement....373
12.4.3 Using Arrays for Dispatch....374
12.5 When to Use Which Construct....376
Chapter Summary....378
Exercises....380

Table of Contents

Chapter 13 Encapsulation **383**
13.1 Design, Abstraction, and Encapsulation 384
13.2 Procedural Abstraction 385
13.2.1 The Description Rule of Thumb 385
13.2.2 The Length Rule of Thumb 387
13.2.3 The Repetition Rule of Thumb 387
13.2.4 Example 388
13.2.5 Benefits of Abstraction 389
13.3 Protecting Internal Structure 390
13.3.1 private 391
13.3.2 Packages 391
13.3.2.1 Packages and Names 392
13.3.2.2 Packages and Visibility 394
13.3.3 Inheritance 396
13.3.4 Clever Use of Interfaces 398
13.4 Inner Classes 398
13.4.1 Static Classes 399
13.4.2 Member Classes 399
13.4.3 Local Classes and Anonymous Classes 402
Chapter Summary 407
Exercises 408

Chapter 14 Intelligent Objects and Implicit Dispatch **409**
14.1 Procedural Encapsulation and Object Encapsulation 410
14.2 From Dispatch to Objects 412
14.2.1 A Straightforward Dispatch 412
14.2.2 Procedural Encapsulation 413
14.2.3 Variations 414
14.2.4 Pushing Methods Into Objects 416
14.2.5 What Happens to the Central Loop? 417
14.3 The Use of Interfaces 418
14.4 Runnables as First Class Procedures 422
14.5 Callbacks 424
14.6 Recursion 428
14.6.1 Structural Recursion 428
14.6.1.1 A Recursive Class Definition 430
14.6.1.2 Methods and Recursive Structure 431
14.6.1.3 The Power of Recursive Structure 432
14.6.2 Functional Recursion 434
Chapter Summary 436
Exercises 437

Table of Contents

Chapter 15 Event–Driven Programming....................439
15.1 Control Loops and Handler Methods....................440
15.1.1 Dispatch Revisited....................440
15.2 Simple Event Handling....................443
15.2.1 A Handler Interface....................443
15.2.2 An Unrealistic Dispatcher....................444
15.2.3 Sharing the Interface....................446
15.3 Real Event–Driven Programming....................448
15.3.1 Previous Examples....................448
15.3.2 The Idea of an Event Queue....................448
15.3.3 Properties of Event Queues....................450
15.4 Graphical User Interfaces: An Extended Example....................451
15.4.1 java.awt....................451
15.4.2 Components....................452
15.4.3 Graphics....................453
15.4.4 The Story of paint....................453
15.4.5 Painting on Demand....................455
15.5 Events and Polymorphism....................456
Chapter Summary....................458
Exercises....................459

Chapter 16 Event Delegation and java.awt....................461
16.1 Model/View: Separating GUI Behavior from Application Behavior....................462
16.1.1 The Event Queue, Revisited....................463
16.2 Reading What the User Types: An Example....................465
16.2.1 Setting Up a User Interaction....................465
16.2.2 Listening for the Event....................467
16.2.3 Registering Listeners....................469
16.2.4 Recap....................469
16.3 Specialized Event Objects....................470
16.4 Listeners and Adapters: A Pragmatic Detail....................472
16.5 Inner Class Niceties....................474
Chapter Summary....................476
Exercises....................477

Part 5 Systems of Objects....................479

Chapter 17 Models of Communities....................481

Chapter 18 Interfaces and Protocols: Gluing Things Together....................503

Table of Contents

Chapter 19 Client–Server Interaction Patterns **525**
19.1 What Is a Client–Server Interaction? 526
19.2 Postal Services: An Example 526
19.2.1 A Server Can Provide a Variety of Services 527
19.2.2 You Can Have More Than One Provider of a Given Service 528
19.2.3 Services Can Be Layered 529
19.2.4 Roles Are Relative to a Service 529
19.3 Implementing Client–Server Interactions 530
19.3.1 Client Pull 531
19.3.1.1 Locating the Server 531
19.3.1.2 Client Pull Tradeoffs 532
19.3.2 Server Push 533
19.3.2.1 Registering with the Server 533
19.3.2.2 Server Push Tradeoffs 533
19.4 The Nature of Duals 534
19.5 Pushing and Pulling Together 535
19.5.1 Passive Repository 536
19.5.2 Active Constraint 537
Chapter Summary 539
Exercises 540

Chapter 20 Synchronization **541**
20.1 An Example of Conflict 542
20.2 Synchronization 542
20.3 Java's synchronized Declaration 543
20.3.1 Synchronizing Methods 543
20.3.2 Synchronizing Blocks 543
20.4 What Synchronization Buys You 544
20.5 Safety Rules 544
20.6 Deadlock 546
20.7 Obscure Details 546
20.7.1 Synchronization and Local Copies of State 547
20.7.2 Synchronized Blocks and Lock Object References 547
Chapter Summary 549
Exercises 550

Chapter 21 Network Programming **551**
21.1 A Readable Writeable Channel 552
21.1.1 Tin Can Telephones 552
21.1.2 Streams 554
21.2 Using a Channel 555
21.2.1 Streams for Writing 555
21.2.1.1 Flushing Out the Stream 555

Table of Contents

Chapter 21 Network Programming

21.2.1.2 A Scribe Example....557
21.2.2 Streams for Reading....559
21.2.2.1 Reading and Blocking....559
21.2.2.2 A Lector Example....560
21.2.3 Encapsulating Communications....562
21.3 Real Streams....564
21.3.1 Abstract Stream Classes....564
21.3.2 Decorator Streams....565
21.3.3 Stream Sources....567
21.3.4 Decoration in Action....568
21.4 Network Streams: An Example....568
21.4.1 Starting from Streams....569
21.4.2 Decorating Streams....570
21.4.3 Sockets and Ports....570
21.4.4 Using a Socket....571
21.4.5 Opening a Client−Side Socket....572
21.4.6 Opening a Single Server−Side Socket....572
21.4.7 A Multi−Connection Server....574
21.4.8 Server Bottlenecks....574
Chapter Summary....576
Exercises....577

Index....579

Part 1

Introduction to Interactive Program Design

Chapter 1

Introduction to Program Design

Chapter Overview

- What is a computer program?
- What are the parts of a program? How are they put together?
- What kinds of questions does a program designer ask?

In this chapter you will learn how a computer can be controlled by a set of instructions called a ***program***. This chapter introduces two different aspects of computation: ***single−minded instruction−following*** and ***coordination among instruction−followers***. The programs in this book involve both aspects of computation.

The first aspect of computation is as step−by−step instruction−following, like the process of making a single sandwich. This kind of computation is a sequence of instructions that produces some desired result. The question that drives this part is *"What do I do next?"* Pieces are put together using instructions like *"Next...,"* *"If...then...else...,"* and *"Until..."* This kind of computation has an end goal that execution of these instructions will accomplish. The programs in this book use short sequences of instructions, executed over and over, to create entities that can provide services or respond to requests (e.g., a sandwich−maker).

The second aspect of computation involves coordinating among many of these instruction−following entities. This is like gathering the sandwich−makers (and table−waiters and others) together to run a restaurant. This kind of computation is

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

creating (and managing) a community. The driving questions are *"Who are the members of the community?," "How do they interact?,"* and *"What goes inside each one?"* The members of the community — the instruction−following entities — are glued together through their interactions and communications. Executing this kind of computation provides an ongoing program such as your car's cruise control, a web browser, or a library's card catalog.

When you finish this chapter, you will know the basic questions to ask about every computational system. These questions will allow you to begin to design a wide variety of computer programs.

Objectives of this Chapter

1. To understand the first aspect of computation: step−by−step instruction−following, like the process of making a single sandwich.

2. To learn (informally) some of the programming primitives for such instruction−followers.

3. To understand the second aspect of computation: constituting a community of interacting entities and coordinating their interactions.

4. To learn the questions that drive the design of a program.

5. To introduce the development cycle of a program.

6. To introduce the role of an interactive control loop.

1.1 Computers and Programs

Computers provide services. A suitably equipped computer can retrieve a web page, locate the book whose author you're thinking of, fly an airplane, cook dinner, or send a message to your friend half way around the world. In order for a computer to do any one of these things, two things must happen. First, the computer must be *told how to provide the required services.* Second, the computer must be *asked to do so.*

The how−to instructions that enable computers to provide services are called ***programs***. A computer program is simply a set of instructions in a language that a computer can be made to follow. When the computer actually follows the program instructions, we say that it is ***executing*** that program. The program is like the script for a play. It contains instructions for how the play should go. But the script itself is just a piece of paper: no actors, no costumes, no set, no action. Executing a program is like performing the play.

Now there is something to watch.

This analogy goes further, too. The same script can be performed multiple times, just as the same program can be executed again and again. If audience reaction (or the director's interpretation, or the theater, or the time of day) influences the performance, two performances of the same script may be quite different. Similarly, user input, hardware, software, or other environmental circumstances may make two different executions of the same program quite different from one another. (Think of running the same word processing program on two different occasions; the experiences are extremely different even though the computer follows the same general−purpose instructions both times.)

When you sit down at a computer, someone else has already told it how to do a lot of things. For example:

- When you press the power switch, it ***boots up***, i.e., gets started running, in the way that it has been instructed to. Personal computers typically come with a fairly sophisticated set of startup instructions already installed. Simply turning on the computer causes the computer to execute this startup program.

  [Footnote: Starting a computer is called ***booting it up***, presumably from the phrase “pulling yourself up by your bootstraps.” The startup program that a computer executes each time that it is turned on is called the computer's ***boot sequence***.]

- Each computer has a program that it runs automatically. The program that your desktop or laptop PC runs is called its ***operating system***.

- A ***disk drive*** — which is really a separate computer plus the electronic equivalent of a huge filing cabinet — comes equipped with instructions for how to retrieve information from (or store information in) that filing cabinet plus how to transmit that information across the cable that connects the disk drive with your “main” computer.

- A microwave oven comes with a computer that follows instructions for how to tell time and how to turn on its microwave generator for specified periods.

- The library's card catalog provides lookup services.

- Your car's cruise control accelerates and decelerates to keep your car moving at a steady rate.

- A ***web browser*** fetches and displays information that it retrieves (at your request) from the hard drives (file cabinets) of computers scattered around the world, with the assistance of the “***web server***” programs running on those distant computers as well as the network (transmission) services provided by a set of intervening

computers.

When you load a new piece of software onto your computer — a cool new game, for example — what you are actually doing is giving your computer a copy of the program: the set of instructions that tells it how to do display graphics and make appropriate sound effects or whatever it is that the particular piece of software does. Writing down these instructions was the job of the person (or people) who wrote the software, the ***programmer***. Loading the software makes the instructions (the script) available to your computer. Just having these instructions lying around doesn't do you much good, though. To actually play the game (perform the play), you need to do one more thing. You need to run the program.

[Footnote: Some computer games can be run off of removable media, like CD−ROMs. In this case, you don't need to load the program onto the computer, but you do need to make sure that the disk is in the drive, i.e., that the instructions are available to the computer.]

Tomorrow, if you want to play the game again, you only have to run it; you don't have to start by loading it onto your computer.

1.2 Thinking Like a Programmer

A computer program — "how−to" instructions for your computer — must be written in a language that the computer can follow. There are many languages designed for instructing computers. These languages are called ***programming languages***, and they are typically quite different from the kinds of languages in which people talk to one another. One of the main differences between talking to a person and programming a computer is the increased level of precision required to tell a computer how to do things. With people, it is often possible to give very vague instructions and still get the behavior you want. A computer has no common sense. You must be very specific with it. Your instructions must be step−by−step, in great detail. In some ways, programming a computer can be a lot like talking to a very young child or a creature from a different planet.

Imagine teaching a Martian how to make a peanut butter and jelly sandwich. You need to give detailed, step−by−step instructions:

1. Get a loaf of bread.
2. Remove two slices of bread and put them on the counter.
3. Get a jar of peanut butter. Put it on the counter, too.
4. Get a jar of jelly. Put it next to the peanut butter.
5. Get a knife.
6. Open the jar of peanut butter.
7. Pick up a slice of bread.
8. Using the knife, pick up a glop of peanut butter and spread it on the top of the slice of bread.
9. ...

These instructions tell the Martian, in very specific terms, what to do. To follow the instructions, the Martian simply needs to perform each step, one by one, in the order given. As long as each of these instructions is one that the Martian knows how to perform, when the Martian finishes executing this program, the Martian will have a peanut butter and jelly sandwich.

If there is an instruction here that the Martian does not understand, that instruction needs to be rewritten in more detail so that the Martian will be able to execute it. For example, *"pick up a glop of peanut butter"* might require further explanation:

a. Insert the knife blade half−way into the jar of peanut butter.
b. Remove the knife from the jar of peanut butter at a slight angle so that some peanut butter is carried out of the jar by the knife.
c. ...

An instruction that needs further explanation before the Martian (or computer) can execute it is one that we call a ***high level instruction***. We can use high level instructions in our programs only if we can supply additional instructions to explain how to actually execute these higher level instructions.

Although we don't know what instructions Martians are likely to understand, a programmer knows what kinds of instructions are a part of the particular programming language in which she is developing a computer program. In this book, we will use a programming language called ***Java***. As you read this book, you will learn how to think like a programmer and how to write instructions that computers can understand. You will also learn specifically about the kinds of instructions that are part of the Java programming language.

As a programmer, you will design sequences of instructions much like the peanut butter and jelly sandwich instructions. The goal of such a sequence is to get something done, to find an answer or to create something. In order to design a program like this, you will need to repeatedly answer the question, *"What do I do next?"* until you have reached your desired result. In many ways, this approach makes computers seem much like sophisticated calculators. In fact, this is where computers got their start: the word ***computer*** used to refer to people who did (mathematical) computations, and the original mechanical computers were designed to perform these computations automatically.

When you are designing a program, you should ask yourself, *"What do I do next?"* You don't necessarily have to write out all of the basic steps in one long sequence. You can group them together in bigger, more abstract, higher level chunks:

I. Assemble the ingredients.
II. Spread the peanut butter.
III. Spread the jelly.
IV. Put the sandwich together.

V. Clean up.

This is a perfectly good set of instructions. But, as in the case of the Martian who didn't know how to *"pick up a glop of peanut butter,"* these instructions will require further elaboration. A programming language such as Java allows you to make up your own high level instructions like *"Assemble the ingredients"* and then to explain how to do this:

> *"1. Get a loaf of bread. ... 4. Get a knife."*

Your program is complete only when every instruction is either understandable by the computer or further explained in terms that are understandable by the computer. **When you are done asking yourself *"What do I do next?"* , you must then ask *"How do I do each of these things?"* until every instruction of your program is something that the computer knows how to do.**

1.3 Programming Primitives, Briefly

What kinds of things do computers know how to do? Most computers don't know how to make peanut butter and jelly sandwiches. Most computers do know how to manipulate numbers and also other kinds of information, like words. In the Java programming language, you will find tools that let you send messages to other computers on a network or create windows and buttons to communicate with people using your programs. Other computers may have special kinds of instructions. A robot control system has instructions that tell the robot when, where, and how to move. A security system may have an instruction to sound an alarm. These are the basic instructions out of which programs for each of these systems can be constructed.

These basic instructions can be combined by ***sequencing*** them, as we've already seen. They can also be ***grouped into mini−programs and given names***, like *"Assemble the ingredients."* These names can then be used as new instructions. When the computer needs to execute one of these new instructions, it simply looks up the rule for how to do it. For example, when the Martian needs to assemble the ingredients, it uses the detailed instructions: *"1. Get a loaf of bread. ... 4. Get a knife."*

Instructions can also be combined in other ways. Sometimes, there is a ***choice*** to be made. For example, after spreading a glop of peanut butter on top of the bread (step 8 in the original list of instructions), the next step in the peanut butter and jelly program might say:

9. If the top of the slice of bread is covered in peanut butter, go to step 10. Otherwise, go back to step 8.

This step contains a choice; the next step might be 8 or it might be 10, depending on whether the slice of bread is full. The Martian (or computer) executing this program will have to keep track of which step comes next. This kind of choice step is called a

conditional, and it is a common construct in programming languages. It is especially useful when the answer to the question *"What do I do next?"* depends on something you won't be able to figure out until you're executing the program.

We might want to go further, replacing steps 8 and 9 with a new kind of step that says

8. Repeat the following substeps until the top of the slice of bread is completely covered in peanut butter:
    a. Pick up a glop of peanut butter.
    b. Spread it on the top of the slice of bread.

This step (*"repeat until"*) is called a ***loop***. It, too, is a common construct in programming languages. Some loops tell you to keep going until something is true (like the bread becoming full), while others tell you how many times to do the steps inside the loop. Some loops even go on forever. For example, a clock is basically a loop that moves its hand(s) (or changes its display) once a minute. Loops are especially useful when part of *"What do I do next?"* is to repeat (almost) the same thing several times.

Each of the techniques described above — sequencing steps, conditionals, loops, and grouping steps into new basic steps (also called *procedural abstraction*) — is an important part of building computer programs. You will learn more about how to do these things in Part 2 of this book. These are the pieces that a programmer uses to answer the questions *"What do I do next?"* and *"How do I do each of these things?"* But this is only one part of the programming problem. **The second part of programming is coordinating the activities of many interdependent participants in a computational community.**

1.4 Ongoing Computational Activity

Some computer programs are very much like peanut butter and jelly sandwich making instructions. They start with some ingredients and step−by−step calculate whatever it is they're designed to create, producing an answer or result before stopping. The original mechanical computers, which mimicked human computers performing mathematical calculations, were very much like this. Sometimes, you would bring your program to a computer operator and then come back the next day for the result!

Today, most computer programs aren't like this. Instead, computer programs today are constantly *interacting*. They may interact with people, machines, other computers, or other programs on the same computer. For example, a word processing program or spreadsheet waits for you to type at it, then rearranges things on the page or recalculates values as you type. A video game moves things around on your screen, some in response to you and others by itself. A web browser responds to your requests, but also talks to computers all across the network. The cruise control system for your car responds to road conditions, sensor readings, and your input. A robot control system interacts with the robot and, through the robot, with the robot's environment, perhaps with no human input

at all.

These computations aren't concerned with solving some pre−specified problem and then stopping. Most computations of interest these days are things called ***servers*** or ***agents*** or even just ***applications***. Most of them have some basic control loop that responds to requests or other incoming information continually. These computations are *embedded in an environment* and they *interact with that environment*: users, networks or other communication devices, physical devices (like the car), and other software that runs at the same time.

These programs are not just interacting with the things around them, either. In fact, each of these programs may itself be composed of many separate pieces that interact with each other (as well as with the world outside the program). **Coordinating the activity among the many entities that make up your program — and their interactions with the world around them — is the second aspect of computer programming.**

This is kind of like taking a group of Martians and organizing them to run a restaurant. Some of the Martians will take orders from and serve food to the customers. Other Martians will need to cook food for the customers. Still others will need to check on supplies, make change, or coordinate other aspects of the restaurant's operation. Each of these Martians will provide services to and make request of other Martians (or to the restaurant's customers or suppliers or other parts of the environment in which the restaurant is embedded). Coordinating the interactions among these Martians (and between the Martian restaurant and its environment) involves different kinds of questions from the instruction−following *“What do I do next?”*

Before we turn to the coordination of activity, though, let's look closely for a moment at one of the Martians who will staff our restaurant. We will see that, deep down, peanut butter and jelly programming still has an important role to play in creating computational activity. Keep in mind that this Martian represents just one of the many things going on in our restaurant.

The instructions that a Martian chef follows might look very much like this:

1. Pick up a new food order.
2. Find the instructions for the dish ordered and follow them.
3. Put the completed dish and the order information on the counter for pickup.
4. Go back to step 1.

Step 2 of this program is the kind of “higher level” step that we described above. It is not itself complete; instead, it refers to other, more detailed instructions to be followed. For example, if an order comes in for a peanut butter and jelly sandwich, the Martian chef will need to use the instructions developed above for how to make a peanut butter and jelly sandwich. A computer still follows simple sequenced steps written in a language that it can execute. But while this Martian is making a peanut butter and jelly sandwich,

another Martian is asking the customer at table 3 whether she would like some more water. Later, the Martian waiter will come into the kitchen and pick up the sandwich that the Martian chef just made. And when the Martian chef is done making the peanut butter and jelly sandwich, the Martian will turn to the next food order, continuing its ongoing interaction.

The peanut butter and jelly style of program instructions is an important part of how the Martian chef does its job. But the Martian chef's instructions are not simply the steps of the peanut butter and jelly program. The basic structure of the Martian chef program is an ***infinite loop*** — a loop that goes on forever. This program accepts requests (in the form of new food orders) and provides services (in the form of the completed dishes) over and over again. We sometimes call this kind of loop — one that provides the main behavior for a participant in the interactive program community — its ***control loop***. Many program community participants take this form, and we will look more closely at control loops in Part 3 of this book. Programs with ongoing central control loops like this are the members of our interactive computational community.

1.5 Coordinating a Computational Community

At its most basic level, every computer program is made of instructions that are followed, one by one. But a single computer program may have many instruction−followers inside it, just as our restaurant is run by many individual Martians. When you look at the whole program — like the whole restaurant — you don't necessarily see the individual instruction steps. Instead, you see coordinated activity among a group of interacting entities. The behavior of this community — providing customers with hot meals — is not the responsibility of any particular member of the community. Instead, it is the result of many community members working together in a coordinated fashion.

Building modern interactive software involves something very much like ***organizational design***. We call this part of programming ***"constituting a community of interacting entities."*** The programmer's job is to figure out how to tell the computer what to do, and no matter what the specific problem to be solved may be, there are fundamental questions that each programmer must ask. Designing a computation which is a community of interacting entities involves figuring out who the members of this community are, how each one works, and how they interact. This is like setting the cast of a play, or deciding what the sub−units of your business will be, as well as how they should interrelate. In planning the organizational structure of your business (or program), you also have to figure out how each unit works and what — and how — they are supposed to communicate. These are the big questions of this second aspect of programming.

When you are designing this kind of activity, you ask yourself several questions:

- What is the desired behavior of the program?
- Who are the entities who interact to produce this behavior?

- How do these entities interact?
- What goes inside each entity (how does it work)?

In the remainder of this section, we will expand these questions and begin to explore them in somewhat greater detail. Understanding these questions and their ramifications is the theme of this entire book. Coordinating communities is a special focus of Part 4.

1.5.1 What Is the Desired Behavior of the Program?

Before you can design a system to solve your problem, you must know what your problem is. This involves knowing not only what you want, but how it should work or fail to work under a variety of different circumstances.

Some questions that you ought to be able to answer about your desired program include:

- What services should your program provide?
- What guarantees does your program make about these services?
- Under what assumptions (circumstances, conditions) does your program make these guarantees?

Consider the restaurant of the previous section. What can we say about its behavior? In answering this question, we consider both the experiences of individual customers and the ongoing properties that the restaurant must maintain, such as remaining solvent. A basic ***specification*** of the service provided by the restaurant might be: Each customer is seated at a clean table, the order is taken, food is served, a bill presented, and payment collected.

There are a number of guarantees we want to make about these services. For example, customers should not have to wait for an unduly long time. Different parts of the restaurant must communicate; customers should not be charged for food that they were not served, etc. Over time, the restaurant should take in at least enough revenue to cover its operating expense. Supplies should not run out, nor should they rot.

We will make certain assumptions in order to be able to provide these guarantees. For example, the "timely service" guarantee will be possible only if the load on the restaurant is reasonable. We might decide that we will be able to uphold this guarantee only if the number of people wanting to eat in the restaurant at one time never exceeds its capacity and if the rate of arrival of these people doesn't exceed the rate at which the restaurant can serve them.

[Footnote: How many customers the restaurant can handle at the same time is called its ***bandwidth***. How quickly each one can be served is called its ***latency***. The number of

customers per hour that the restaurant can handle is its ***throughput***. These quantities — bandwidth, latency, throughput — are common measures of program performance.]

These assumptions should be made explicit, and we will also need to say what happens when they are violated. (In this case, the timely service guarantee won't be upheld, but how slow the service gets should be related to how overloaded the restaurant is.)

There are other assumptions we do not make about our program, and we can articulate these as well. We do not assume that only one customer will be served at a time. Instead, we expect that multiple tables must be handled (roughly) simultaneously. It certainly won't do to wait until the first has eaten, paid, and left before addressing the second. We also permit different interactions with each table to be handled simultaneously or at least overlapped: food may be cooking while checks are being written up.

This description is still fairly general, and we can imagine making it more specific. (For example, are customers constrained to ordering off of a menu?) In general, the more detail you can give of what your program ought to do, the easier your task will be in designing and building it.

1.5.2 Who Are the Entities Who Interact to Produce the Program's Desired Behavior?

This question can't be answered in isolation, because any and every decision you make about *who* the entities are is also at least a partial commitment to *what* they are and *how* they work. So answering this question is in many ways like solving the whole problem. The trick is to answer this “who” question in fairly high−level, general terms, then to sit down and try to hash out the answers to all of the “how they interact” and “how they work” questions. In answering those, you'll almost certainly have to return to this “who” question and rearrange your answer a few times. This is fine; it's even typical enough to have a name: ***incremental program design***.

In the restaurant, an appropriate high level division of labor might have a wait staff unit (the people who deal directly with the customers), a kitchen staff unit (the people who cook the food), and a financial unit (who keep track of how much which things cost, collect money, and buy supplies). At this point, we haven't committed to whether these are three roles played by a single Martian, three separate Martians, or even three groups of several Martians each.

1.5.3 What Goes Inside Each Entity (How Does It Work)?

To answer this question requires knowing a bit about how each entity will interact with the other members of its community. This means that answering *“How does each entity work?”* is closely related to *“How do they interact?”* After all, specifying what interactions each entity needs to support goes a far way towards telling you whether the answers to the *“How does each entity work?”* questions meet the requirements of the

community.

Some subsidiary questions to ask in order to determine how each entity works include:

- What responsibilities does it have?
- What guarantees (promises, commitments) does it make? Under what assumptions?
- What resources does it control?
- Is it a community, too?

For example, the restaurant's wait staff might be responsible for greeting the customers in a timely fashion, supplying each one with a menu (a structure that the program will have to provide and keep updated!), taking the order, delivering it to the kitchen staff, picking up and serving the cooked meal, obtaining a price from the accounting entity, and obtaining payment for that amount from the customer. The wait staff might guarantee to communicate with (most of) the customers within minutes, provided the total number of customers is limited and the maximum time spent with each is under a certain amount. It might also promise to deliver food within some small amount of time after it's done cooking, provided that the kitchen staff notifies the wait staff in a timely manner. The wait staff controls menus, knows which food items were ordered by which customers, and is the only part of the restaurant that deals directly with the customers. And so on.

When it comes to *“What goes inside each entity (how does it work)?”,* there are two kinds of answers. One answer is that the behavior of the entity is accomplished by a single rule−follower running an interactive control loop. We saw an example of this when we considered the Martian chef earlier. In this case, we ask “What does the Martian do next?” over and over, until we wind up with a well−defined set of instructions for this Martian to follow.

The other possible answer to the question *“What goes inside each entity (how does it work)?”* is that this entity is itself a community. (The wait staff might be further divided into the person who takes the order, the person who clears the table, and the person who serves the wine.) In this case, we need to figure out how to build each of these entities, asking again *“What goes inside each entity (how does it work)?”* The problem of figuring out how to coordinate the activity of a community continues until each community member is a single (rule−follower) Martian. Then we ask about the instructions that this Martian follows.

1.5.4 How Do These Entities Interact?

This question concerns coordination and communication among two or more entities. Some of the questions that you should ask about how these entities interact include:

- What are the entities' interfaces?
    - ♦ What promises does each one make?
    - ♦ What contracts does it fulfill?
    - ♦ What services does it provide?
- How do they communicate?
    - ♦ What mechanisms do they use?
    - ♦ What interaction patterns do they use?
    - ♦ How do they preserve ***liveness***, i.e., how do they make sure that things keep moving?
- What interaction patterns are possible?
- What happens when something goes wrong?

A ***protocol*** is the specification for an interaction between two entities. For example, a common protocol for the interaction between the wait staff and kitchen staff of a restaurant involves a slip of paper with the customer's order written on it. The waiter hangs this piece of paper in the window over the kitchen's food pickup counter, a place where it will be easy to find when someone from the kitchen is ready for a new job. When a member of the kitchen staff is ready to process the order, the piece of paper is removed and used to guide the food preparation. When the order is ready, it is placed on the food pickup counter together with the original order slip. This identifies the food with the original request when the waiter returns to retrieve it. The slip of paper serves as a crucial reminder of several associated pieces of information: what was ordered, by whom, and where they are seated.

Protocols can also address temporal issues. For example, the wait staff/kitchen staff interaction described in the preceding paragraph needs to happen in ***real time***, meaning that the protocol itself can't introduce significant delays. There must also be guarantees made about the frequency with which the wait staff checks for completed dishes (or the kitchen staff for incoming orders). If assumptions such as these are built into protocols, they must be documented so that they are maintained in the behavior of participant entities.

In contrast, the wait staff interacts with the financial unit by obtaining prices for food and turning over any moneys collected. These interactions could happen in ***batch***, meaning that it is OK for the wait staff to get the price list at the beginning of the week or for money to be handed over at the end of the day.

[Footnote: Batch processing is like the old−fashioned computations in which you handed your program to a computer operator and came back the next day for your results.]

The difference between real time and batch interactions is only one dimension that must be determined in order to coordinate the activities of the members of your computational community.

A protocol specifies the interface, or meeting, between various entities in the community that constitutes your program. Once the interfaces have been thoroughly fleshed out, each entity can in theory be implemented by a separate programmer (or team of programmers) provided that it is built to spec, i.e., that it meets the specifications of the agreed−upon interface.

In practice, the task of implementing an entity to match a given specification often results in questions about or revision of that interface. Programming is not so neat a task as students of computer science would often like to believe; there's a cycle of specification and implementation, debugging and testing, usage and revision, that characterizes almost all real−world software. The later stages of this process are sometimes called the ***software life cycle***; but the repeated revision that characterizes those later stages start before a piece of software is even born.

1.6 The Development Cycle

The sections above concern the ***design*** of a computer program. Typically, you will be given a set of specifications and some components that need to be integrated into the system you build. Perhaps you will only be asked to build a single entity or to modify existing entities to facilitate coordination. Regardless of your particular design problem, you will find it useful to situate your task in the context of these six questions:

- What is the desired behavior of the program?
- Who are the entities who interact to produce this behavior?
- How do these entities interact?
- What goes inside each entity (how does it work)? In particular, is each one made of a community of entities or a single instruction−following control loop?

And, when we get down to instruction−followers:

- What does it do next?
- How does it do each one of these things?

Once you have the answers to all of these questions, you can start to build your program. Of course, you will already have found that you needed to go back to earlier parts of the design process to modify or flesh out various decisions. You may also have shown your completed design to other programmers — or, perhaps more importantly, to the users or customers for whom you are creating this service — and revised your design specification in response to their feedback.

The ***implementation*** phase of the project is no different. In building a program that is supposed to meet your specification, you will often find that you need to go back and change that specification. When this happens, you need to be careful to consider all of the interdependencies that led you to your original design. That is, **the development of software is cyclic,** beginning with design but often returning to it. It will not always be desirable (or even possible) to change your design, but it is quite common to discover additional assumptions or nuances that must be percolated through the design during later phases of development.

When you begin to build your program, it is often advisable to implement only a small piece of your system first. This may mean implementing only some of the entities, or it may mean implementing all of the entities but only simple, basic versions of each. In large scale system development, this initial phase is called ***prototyping***. Even in most of the smaller scale programs that you will encounter in your early coursework, it is a good idea to utilize this approach of ***incremental program development***. Part of developing good programming skills involves learning to consciously and explicitly design a staged development plan in which smaller simpler programs are constructed and tested, then gradually expanded until the desired functionality is obtained.

Building a simpler version of your system gives you an opportunity to test your basic approach before you have built up too much complexity. It also means that your ***bugs***, or ***program errors***, will be easier to find. Bugs come in many flavors, ranging from simple syntactic errors such as spelling mistakes, to programming errors such as incorrect variable scoping, to conceptual design problems such as impossible−to−meet but critical guarantees.

Even after you've found the bugs that keep your program from running, you will need to subject your code to rigorous ***testing***. This means trying out not only the “normal,” expected behavior, but also checking how your program handles unexpected or anomalous behavior. Think of your program as an opponent you're trying to trick; see if you can get it to misbehave. This testing — when done right — will lead you to modify your code or even your design.

This repeated cycling through and between the various stages of specification (or design) development, implementation, and testing is a crucial skill for any good programmer. Classroom programs are too often written once and tested on obvious cases. Most of the time and money spent on real−world software is spent on revision and maintenance rather than on initial development. Acquainting yourself with this cycle — and with

writing clean, easy−to−read, reusable code — may be the most important part of becoming a skilled programmer. These issues — together with a tour through the development cycle — are the topic of the next chapter.

1.7 The Interactive Control Loop

This book focuses on the problem of designing interactive software. At the heart of our approach is the idea of an ***interactive control loop***. This is a simple program (set of instructions) that repeatedly receives an input — a new request, a set of sensor readings, or some other information — and responds appropriately. In the general case, the response may involve initiating a series of other activities, so this kind of program can in principle become almost arbitrarily complex. The basic idea is rather simple, though.

To conclude this chapter, we present an extremely simple interactive control loop. This example will be used as a motivator for the development of the next part of the book. The interactive control loop idea is a theme that runs through this entire book. In a way, it might be thought of as the "atomic unit" or basic vocabulary element of this kind of computation.

Perhaps the simplest interactive control loop is an ***echo program***. When run, this program waits for the user to type something. When the user finishes typing, the program simply repeats back what it has been given. That is, it's a loop that gets some input, processes that input (in this case trivially), and then spits out its result.

Although the echo program seems too trivial to be of much use, a minor variant of it runs in almost every program you type to: it's what makes the characters appear on the screen. Far more importantly, the basic structure of this program underlies essentially every interactive computation. And it demonstrates many of the important properties of an interactive computation:

- It is ***embedded in an environment*** (in this case involving a user's typing and a display that the user can see).
- It is ***interactive*** (with that user, but we could have it talk to another program or over a network instead).
- It is ***concurrent***: other things happen at the same time that the program is running. (In this case, the user might be typing the next line even while the echo program is producing its output.)

The idea of an interactive control loop is the root of this approach to programming. By putting together interactive control loops, you constitute a community of interacting entities. Interactive control loops are what goes inside; communication between them is how they interact. In other words, as they say, all the rest is corollary...

Chapter Summary

- Computers follow special instructions, called a *program*, which is written in a special *programming language*.
- Computation results when a computer has access to these instructions and *executes* them.
- Each set of instructions must answer:
    - What should the program do next?
    - How should it do it?
- Groups of steps can be combined to make a *"higher order"* step.

  This is called *procedural abstraction*.
- Steps can involve *choices* or *decisions*.
- Steps can be executed over and over again using a *loop*.
- Most modern programs combine many separate looping *instruction−followers* into an *interacting community*.
- Every computation is embedded in an environment and interacts with the other (computational and physical) entities around it.
- The programmer's job in designing the program is to figure out:
    - What is the desired behavior of the program?
    - Who are the entities who interact to produce this behavior?
    - How do these entities interact?
    - What goes inside each entity (how does it work?)
- Program construction is a cycle of designing, building, testing, and then designing again.

Exercises

1. Give step−by−step instructions for how to tie shoelaces.

2. Select your favorite recipe and give step−by−step instructions for how to cook it.

3. Give detailed directions for how to get from your classroom to where you live. Include indications that will tell whether you've gone too far and how to get back on track.

4. Specify the expected behavior for each of the following interrelated services provided by a bank account:

    a. A deposit.
    b. A withdrawal request.
    c. Checking your balance.

    Does your specification permit overdrafts?

5. You are at a fruit market. Describe the protocol by which you purchase a piece of fruit from the fruit seller.

6. You are at a yard sale and see an old but comfortable−looking upholstered chair that you are interested in buying. Describe the protocol by which you negotiate for and (possibly) buy the chair.

7. Describe the division of responsibility and coordination of activities among the entities at a major airport.

   In doing so, address the four key questions:

   a. What is the desired behavior of the system?
   b. Who are the entities who interact to produce this behavior?
   c. How do these entities interact?
   d. What goes inside each entity (how does it work)?

   Your answer need not be complete but must address each of the above questions at least somewhat. Your answer should occupy about one−half to one page, single−spaced.

   For the third and fourth of the four questions above, pick just two or three interacting entities to discuss.

   As with all written work for this course, your answer must be legible, should be clear yet concise, and should avoid distracting errors like spelling errors.

8. Proceed as in the previous question, except this time:

   Describe the division of responsibility and coordination of activities among the entities at a major−league baseball game.

9. Proceed as in the previous question, except this time:

   Describe the division of responsibility and coordination of activities among the players on a soccer team.

Chapter 2

The Programming Process

This chapter has not yet been written.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

Interlude A

A Community of Interacting Entities

Chapter Overview

- What are the basic concepts of Java programming?

This interlude provides a whirlwind introduction to most of the basic concepts of Java programming. It uses a simple community of word games and other String transformers to illustrate this exploration.

This interlude is not intended to be read as standalone coverage of these ideas. Instead, it introduces many concepts only briefly, but in context. Each of the programming concepts presented here is reintroduced in much greater detail in the chapters of Part 2.

Objectives of this Chapter

1. To increase familiarity with the design process.
2. To understand how to describe a system design in terms of types, components, and interactions.
3. To discover how design translates into executable code.
4. To be able to read and begin to understand fragments of Java programs.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

A.1 Introduction: Word Games

When I was a child, we used to amuse ourselves by speaking to one another in a special language called ***Pig Latin***. The simplest version of Pig Latin has just one rule: To turn an English word into a Pig Latin one, you take the first letter off the word, then add the first letter plus "ay" to the end of the word. So, for example, "Hello" in Pig Latin is "ello−Hay", and "How have you been?" is "ow−Hay ave−hay ou−yay een−bay?" There are more sophisticated rules for Pig Latin that deal with consonant blends and words that begin with vowels, but the basic idea remains the same. It turns out that there are children's games like Pig Latin in many, many languages, though each has a slightly different set of rules. Another such game, popularized by the children's Public Television show Zoom, is ***Ubby Dubby***, in which you add "ubb" before every vowel (cluster): "Hubbellubbo", "Hubbow hubbave yubbou bubbeen?"

This interlude explores such word and phrase transformations. In fact, we're going to build a system in which you can have many of these different Transformers, and you can glue them together in almost any order. In this sense, the Transformers will be interconnectable modules like Lego™ or Capsela™.

Picture of Transformers

In addition to Transformers such as Pig Latin and Ubby Dubby, we'll want ***Capitalizers*** ("HELLO"), ***Name Droppers*** ("Lynn says Hello", or "Chris says Hello", or "Pat says How are you doing?"), even ***Delayers*** (e.g., that don't produce "Hello" until after they've already received "How are you doing?") or ***Network−senders*** (that can move one of these strings−of−words from one computer to another). We'll also have some community members that can read information that a user types to them or display information on a computer screen. And we'll have Transformers that can listen to two different inputs, producing only one output, as well as Transformers that can produce two outputs from only one input. (The first of these is a ***Combiner***; the second is a ***Repeater***. The first is good when you have lots of people trying to talk all at once; the second is a nice way to circulate (or broadcast) information that needs to get to a lot of people.)

In the system that we're going to explore, we will need a way to create individual transformer−boxes like the ones described above. We'll also need a way to connect them

together. Finally, the transformer−boxes will need to act by themselves, to read inputs, do transformations, and produce outputs. The complete system will be a community of interacting entities, many of which will themselves be communities. At the most basic level, each of these entities will need to follow specific instructions. In this interlude, we will explore both the design of the community and the specific instructions that some of these entities will follow.

A.2 Designing a Community

We need to design:

- What is the desired behavior of the system?
- Who are the entities who interact to produce this behavior? That is, who are the members of the community?
- How do these entities interact?
- What goes inside each entity (how does it work)?

We can start at the bottom (***bottom−up design***) or at the top (***top−down design***). Both are legitimate and useful design techniques. However, in practice design often mixes these techniques. In this case, we're actually going to start in the middle; in this particular system, that is one of the easiest places to begin thinking about what we want to produce.

At the end of the design process, we should be able to sketch out a scenario for each of the major interactions with our system, including what roles need to be filled (i.e., the types of things in our system), who fills these roles (i.e., the individual objects that make up the system), and how they communicate among themselves (i.e., the flow of control among these objects).

A.2.1 A Uniform Community of Transformers

There are several communities implicit in the system that we're building. Let's start in the middle, where the system can be understood as a community of interacting ***Transformers***. In this picture, each Transformer is an entity. The interactions in this community are quite simple: Each Transformer reads in a phrase and writes out a transformed version of it. In this system, we want to be able to interconnect these Transformers in arbitrary ways. This means that the services each Transformer provides will need to be compatible, so that one Transformer can interact with any other Transformer using the same connection mechanism.

Transformer Entity interactions, version 1
♦ Read a word/phrase (from a connection). ♦ Write a word/phrase (to a connection).

We will accomplish this generic connection between Transformer entities using a computer analog of the tin can telephones that we built as children. This is a simple device that allows you to put something in one end and allows someone else to retrieve it at the other end:

Picture of tin−can telephone

Take two tin cans with one end removed from each. Punch a whole in the center of the intact end of each can. With a long piece of string, thread the two cans so that their flat ends face each other. Tie knots in the ends of the string. Pull the string tight, so that it is stretched between the two cans. Talk into one can; have someone else listen at the other.

The computer analog will be ***Connection*** objects that allow one Transformer to write a word or phrase and another Transformer to read it from the Connection.

Picture of Transformers and Connections

The Transformers on either end don't have to know anything about one another; they can simply assume that the Transformers will interact appropriately with the Connection. And the Connections don't have to know much of anything about the Transformers, either:

Connection Entity interactions
♦ Accept a word/phrase written to you. ♦ Supply a word/phrase when requested (read).

Connections provide one particular way of providing interconnections among objects. In this system, the components are designed so that any outputter can be connected to any inputter. In other parts of this book, we will see examples of other kinds of interaction mechanisms. For example, in some systems, the pieces to be interconnected are not uniform. In others, the particular choices of interconnections must be made at the time that the system is designed rather than while the system is running. In Part 4 of this book, we will pay particular attention to the tradeoffs implicit in different interconnection mechanisms.

A.2.2 The User and the System

Before we look at how each Transformer (and Connection) is built, let's step back from this community of interacting Transformers to ask how it came into existence. At this level, the members of our community are the ***user*** who constructs the community and the ***system*** to be constructed. The user expects the system to provide a way to create Transformer entities and a way to connect them.

System/User interactions
♦ Create a Transformer (of a specified type). ♦ Connect two Transformers (in a particular order).

We'll accomplish the first of these by adding another entity to the community: a ***user interface*** containing a Control Panel that allows the user to specify that a Transformer should be created as well as what type of Transformer it should be.

Picture here of Transformer community. [Picture of Control Panel & Transformers.]

The second interaction, connecting Transformers, we will handle by letting the user specify two Transformers (through the user interface) and then asking the specified Transformers to accept a new Connection. So allowing the system to interact with the user creates one additional entity (the user interface):

User Interface interactions
♦ Create a Transformer (of a specified type). ♦ Create a Connection between two Transformers.

and adds an interaction to the Transformer:

Transformer Entity interactions, version 2

- Accept an input Connection (or maybe more than one).
- Accept an output Connection (or maybe more than one).
- Read a word/phrase (from a Connection).
- Write a word/phrase (to a Connection).

[Picture of user interaction flow: click button −> create Transformer,

click Transformers −> create connection & request Transformers accept it.]

Specifically, the Control Panel will have buttons representing each kind of Transformer available. Clicking on a button will create a new Transformer of the appropriate type. Clicking on first one Transformer, then another, will create a Connection between them. This task is actually cooperative: the user interface will create the Connection and it will ask the Transformers to accept it.

A.2.3 What Goes Inside

In the two subsections immediately above, we've designed Transformer−Transformer interactions (via Connections) and user−system interactions (via the user interface). We've addressed the question of who our community members are (user interface, Transformers, Connections, and — stepping back — the user) and, to a first approximation, how they interact. In terms of system design, Transformers and Connections represent kinds of things of which there may be many separate instances. For example, a particular community of Transformers may contain five Transformers and four Connections, or eight Transformers and three Connections, or twelve. Each community will contain only a single Control Panel, though.

The next step in a full design process would be to look inside each of these entities to discover whether they are, themselves, monolithic or further decomposable into smaller communities. We will not decompose the user interface further in this chapter; much of the necessary background for this task will not be introduced until Part 3 of this book. Instead, the remainder of this interlude will look inside the Transformer type to see how these objects are built.

A.3 Building a Transformer

We have seen above the specification of the interactions that a Transformer entity will be expected to fulfill. We can turn this interaction specification around to provide a specification of the behavior that an implementation will need to satisfy: A Transformer must be able to:

- Accept an input Connection.
- Accept an output Connection.
- Have its own instruction−follower that acts independently to read its input, transform that input as appropriate, and write its output.

In fact, this Transformer is itself a community:

Picture here of Transformer community.

The Connection Acceptors are each entities that are activated only on a connection−accept request; their jobs are to remember the Connections that they have been handed. For example, the acceptInputConnection instructions basically say, “To accept an input Connection (let's call it *in*), simply store *in* away somewhere so that you can use it later.” There's also a little bit of additional code that tells the Connection Acceptors what to do if they've already got an input Connection stored away.

Output Connections — another part of the community inside an individual Transformer — are handled in the same way as input Connections. Also, some kinds of Transformers will have entities that need to perform certain initialization activities when an individual

Transformer is created. Finally, the independent instruction−follower is an additional ongoing interacting entity. It makes use of the Connections (such as *in*) that the Connection Acceptors have stored. Each Transformer will have its own instruction−follower, allowing the Transformer to do its work without any other entity's needing to tell it what to do.

For the moment, we will focus on the heart of the Transformer, the work done by this independent instruction−follower, especially the transformation it actually performs. We begin by looking at some specific Transformers and describing the behavior we expect.

A.3.1 Transformer Examples

The instructions for the behavior of a ***Capitalizer*** will say

Capitalizer
1. Read the input. 2. Produce a capitalized version of it. 3. Write this as output.

Every individual Capitalizer is the same, and each one does the same thing. You can tell them apart because they're connected to different parts of the community and are capitalizing different words, though.

NameDropper is a different kind of Transformer. Each individual NameDropper has its own name that it likes to drop. So the instructions for a NameDropper will say

NameDropper
1. Read the input. 2. Produce a new phrase containing your name, the word `"says"`, and the input. 3. Write this as output.

Variations in Transformer behavior aren't restricted to the transformation itself. Yet another kind of Transformer is a ***Repeater***. The repeater is different because it can accept more than one output Connection: two, in fact. The instructions for a Repeater say:

Repeater
1. Read the input. 2. Write this to one OutputConnection. 3. Write this to the other OutputConnection.

And, of course, the instructions for a (simple) ***PigLatin*** should say

PigLatin
1. Read the input. 2. Produce a new phrase containing all but the first letter, then the first letter, then the letters `"ay"`. 3. Write this as output.

As you can see, the basic instructions for a Transformer are of the form:

Basic Transformer
1. Read the input. 2. Produce a transformed version of it. 3. Write this as output.

We will begin by looking at the second of these instructions.

A.3.2 Strings

In Java, there is a special kind of object, called a ***String***, that is designed to represent these words or phrases. In fact, in Java a String can be almost any sequence of characters typed between two double−quote marks, including spaces and most of the funny characters on your keyboard. (The double quotes aren't actually a part of the String itself; they simply indicate where it begins and ends.) For example, legitimate Java Strings include `"Hello"` and `"this is a String"` and even `"&())__)&^%^^"`. (Strings don't have to make sense.) The Transformers that we will build are really StringTransformers, since each one takes in one String at a time and produces a corresponding, potentially new or transformed String as output.

A.3.2.1 String Concatenation

Once you have a String, there are several things that you can do with it. For example, you can use two Strings to produce a third (new) String using the String ***concatenation*** operator, +. In Java,

```
"this is a String" + "%%^$^&&)) mumble blatz"
```

is for all intents and purposes the same as just typing the single String

```
"this is a String%%^$^&&)) mumble blatz"
```

[Footnote: Note that there is no space between the *g* at the end of `String` and the *%* at the beginning of `%%^$^&&))`.]

So, for example, a NameDropper Transformer might use + to create a new String using the input it reads, the name of the particular dropper, and the word `"says"`. Pig Latin and Ubby Dubby might use +, too, but they'll have to pull apart the String they read in first.

A.3.2.2 String Methods

Java Strings are actually rather sophisticated objects. Not only can you do things with them, they can do things with themselves. For example, you can ask the String `"Hello"` to give you a new String that has all of the same letters in the same order, but uses only upper case letters. (This would produce `"HELLO"`.) The way that the String does this is called a ***method***, and you ask the String to do this by ***invoking*** its method. In this case, the name of the method that each String has is *toUpperCase*. You ask the String to give you its upper–case–equivalent by putting a **.** (dot) after the String, then its method name:

```
"Hello".toUpperCase()
```

yields the same thing as `"HELLO"`.

You can also ask a String for a substring of itself. In a String, each character is numbered, starting with 0. (That is, the 0th character in `"Hello"` is the *H*; the *o* is the 4th character.)

[Footnote: Computer scientists almost always number things from 0. This is apparently an occupational hazard.]

So you can specify the substring that you want by supplying the index of the first character of the substring, or by supplying the index of the first character along with the index of the last character plus one. (So the difference between the two indices is the length of the substring.) For example:

```
"Hello".substring(3)
```

is **“lo”** while

```
“Hello”.substring(1, 3)
```

is **“el”** and

```
“Hello”.substring(0)
```

is still **“Hello”**.

These and other useful functions are summarized in the sidebar on *Selected String Methods*.

Selected String Methods

Below are some selected methods that can be invoked on individual Strings, along with brief explanations and examples of their usage.

- ***toUpperCase***() produces a String just like the String you start with, but in which all letters are capitalized. For example:

  ```
  “MixedCaseString”.toUpperCase()
  ```

  produces **“MIXEDCASESTRING”**

- ***toLowerCase***() produces a similar String in which all letters are in lower case. So:

  ```
  “MixedCaseString”.toLowerCase()
  ```

  produces **“mixedcasestring”**

- ***trim***() produces a similar String in which all leading and trailing white space (spaces, tabs, etc.) has been removed. So:

  ```
  “    a very spacey String        ”.trim()
  ```

  is just **“a very spacey String”**

- ***substring***(*fromIndex*) produces a shorter String containing the same characters that you started with, but beginning at index *fromIndex*. Bear in mind that the index of the first character of a String is 0.

substring(*fromIndex*, *toIndex*) produces the substring that begins at index *fromIndex* and ends at *toIndex* − 1. So:

```
"Hello".substring(3)
```

is **"lo"**

```
"Hello".substring(1, 4)
```

is **"ell"**, and

```
"Hello".substring(0)
```

is **"Hello"** again.

- ***length***() returns the number of characters in the String. For example,

```
"Tee hee!".length()
```

is 8. Since the String is indexed starting at 0, the index of the final character in the String is the String's **length() - 1**.

- ***replace***(*old*, *new*) requires two characters, *old* and *new*, and produces a new String in which each occurrence of *old* is replaced by *new*.

[Footnote: A ***character*** is, roughly, a single alphanumeric or symbolic character (one keystroke) inside **single** quotation marks. For more detail on what exactly constitutes a character, see Chapter 3, *Things, Types and Names*, and its sidebar on *Java Primitive Types*.]

For example,

```
"Tee hee!".replace('e', '*')
```

produces **"T** h**!"**

- ***charAt***(*pos*) requires an index into the String and returns the character at that index. Recall that Strings are indexed starting at 0. So:

```
"Hello".charAt(2)
```

returns the same character as **"Hello".charAt(3)**.

- ***indexOf***(*character*) returns the lowest number that is an index of *character*

in the String. So:

```
"Hello".indexOf('H')
```

is 0 and

```
"Hello".indexOf('l')
```

is 2. Also,

```
"Hello".indexOf('x')
```

is −1, indicating that `'x'` does not appear in `"Hello"`.

- ***lastIndexOf***(*character*) returns the highest number that is an index of *character* in the String. So:

```
"Hello".lastIndexOf('H')
```

is 0 and

```
"Hello".lastIndexOf('x')
```

is −1, but

```
"Hello".lastIndexOf('l')
```

is 3.

A.3.3 Rules and Methods

Using the String manipulations described in the previous section and sidebar, we can construct the instructions that a variety of Transformers would use to transform a String. For example, we might write:

```
to transform a String (say, thePhrase),
    return thePhrase.toUpperCase();
```

This rule describes the transformation rule for a Capitalizer. Note that *thePhrase* is intended to stand in for whatever String needs to be transformed. The transformation rule can't operate unless you give it a String. Within the body of the transformation rule, a temporary name (in this case, *thePhrase*) is used to refer to this supplied String. The formal term for such a piece of supplied information is an ***argument***, and the formal term for the temporary name that is used to refer to it is a ***parameter***.

A different transformation rule — this one for a pedantic Transformer that seems to think it knows everything — might say

```
to transform a String (say, whatToSay),
    return "Obviously " + whatToSay;
```

Note that we have chosen a different temporary name to represent the String argument. The parameter name doesn't matter; we can choose whatever (legal Java) name we wish.

[Footnote: Legal Java names are covered in the sidebar on *Java Naming Syntax and Conventions* in Chapter 3, *Things, Types and Names*.]

It can be the same name in every Transformer rule, or different in each one. It is only important that we use the same name in a particular rule both when we're specifying the parameter (in the first line of the rule) and in the body of the rule.

Question: Can you think of another kind of Transformer and write its rule? Remember, it should take a String and produce a String.

The rules as we've presented them aren't really Java code, but they are pretty close. To make them legal Java, we need to add a bit more formality and syntax (notation). The formal name for a rule in Java is a ***method***, just like the String methods — *toUpperCase*, *substring*, etc. — above. Somewhere, someone has provided instructions for how to *toUpperCase* so that you can use that method without worrying how it is done. Here, we are providing the instructions for *transform*, so that someone else can use it.

A definition of Capitalizer's *transform* method might say:

```
String transform(String thePhrase) {
    return thePhrase.toUpperCase();
}
```

Aside from the syntax (the details of which are covered in chapters 6 and 7), the one big difference from the rule specification above is that the method definition begins with the word *String* to indicate that the method will produce a String when it is invoked.

Question: Quick quiz: How would you write the pedantic Transformer's *transform* method?

A.3.4 Classes and Instances

What we just described was how to specify a rule. This rule is the rule used by all Transformers of that particular type. In fact, the rule is really the only thing that distinguishes Transformers of that type from other Transformers. We can describe a type

of Transformer by wrapping the method definition in a bit of code that says it's a type. In Java, a type that provides instructions implementing behavior is called a ***class***.

```
class Capitalizer extends StringTransformer {
    String transform(String thePhrase) {
        return thePhrase.toUpperCase();
    }
}
```

This says that Capitalizer is a type (or class) that is very much like the more general class StringTransformer. Its behavior differs from generic StringTransformers by using the particular *transform* rule contained inside the braces { } that delineate Capitalizer's body.

Pedant is similar:

```
class Pedant extends StringTransformer {
    String transform(String whatToSay) {
        return "Obviously " + whatToSay;
    }
}
```

Question: A class that uses your own Transformer rule should be very much like these. Can you write it?

These classes are descriptions of what a Capitalizer or a Pedant should do. They are not Capitalizers or Pedants themselves, though. They're really more like recipes from which a particular Capitalizer or a particular Pedant can be made. To make a Capitalizer, you use the special Java construction expression

```
new Capitalizer()
```

This “cooks up” a particular Capitalizer using the recipe we just wrote. A Pedant is created similarly, but using a different recipe:

```
new Pedant()
```

If we say it again, we can “cook up” another Pedant:

```
new Pedant()
```

Stepping back, this is exactly what we want the buttons on our control panel to do. Pressing the button marked *Pedantic Transformer* should invoke the expression

```
new Pedant()
```

causing a Pedant to appear on our screen. Pressing it again should invoke it again, making a second Pedant appear. We can connect these two together using other user interface functions. Now, if we send the String `"I'm here!"` through a Connection to the first Pedant, it should send the String `"Obviously I'm here!"` to the second Pedant, and the second Pedant should produce `"Obviously Obviously I'm here!"`.

Question: Connecting a Pedant's output to a Capitalizer's input and supplying the Pedant with `"not much"` will produce `"OBVIOUSLY NOT MUCH"`. What happens if you connect a Capitalizer's output to a Pedant's input?

Question: How about Pedant, then Pedant, then Capitalizer, then Pedant? Then Capitalizer?

A.3.5 Fields and Customized Parts

You can already see from the examples in the previous subsection how one class, or type, can describe many different instances. For example, phrases passed through the first Pedant contain at least one `"Obviously"` at the beginning; phrases passed through the second Pedant will begin with at least two `"Obviously"`s. But to really appreciate the power of multiple distinct instances of a type, we need to look at a type that has ***local state*** associated with each instance. The NameDropper Transformer type is a good example of this.

The transformation rule for NameDropper is

```
to transform a String (say, thePhrase),
    return  my name  + " says " + thePhrase;
```

But *my name* here isn't a parameter. It isn't a piece of information that is supplied to the NameDropper each time the NameDropper performs a transformation, the way that *thePhrase* is. Instead, *my name* is a persistent part of the NameDropper. And it is a part of the particular NameDropper instance, not a part of the NameDropper type. After all, each NameDropper drops its own name.

So where does this name come from? As each individual NameDropper is created, it must be supplied with a name. Then, the particular NameDropper remembers its own name, and when it comes time to *transform* a String, the NameDropper uses its own name.

To do this, we need to create a local storage spot that sticks around between transformations. This is done using a special kind of name that is associated with the NameDropper instance. Such a name is called a ***field***. In this case, we'll use a field called *name*, because that's what it will hold. To make it clear in our code that we're referring to a field, we use a syntax sort of like saying *my name*; we refer to the field using

```
this.name
```

In Java, ***this*** is a way of letting an individual instance say "my own."

So the actual *transform* method for NameDropper should read:

```
String transform(String thePhrase) {
    return this.name + " says " + thePhrase;
}
```

This way, if one NameDropper has the name *Pat* and another has the name *Chris*, then *Pat* would transform the String `"Hello"` into `"Pat says Hello"` while *Chris* would make it `"Chris says Hello"`.

[Picture of Pat and Chris transforming the same String, with field visible.]

This method definition needs to be embedded in a class, of course. We also need to add a bit more machinery to the class to make sure that the name is available when *transform* needs it. The first change is to actually create a place to put the name; the second is to write explicit instructions as to how to create a NameDropper so that it has a name from the very beginning. This second — ***constructor*** — rule will need to say:

```
to construct a NameDropper with a String (say, whatMyNameShouldBe),
    assign my name the value of whatMyNameShouldBe;
```

When we translate this into Java using the special syntax for a constructor rule, it looks like this:

```
NameDropper(String whatMyNameShouldBe) {
    this.name = whatMyNameShouldBe;
}
```

So the whole NameDropper class reads:

```
class NameDropper extends StringTransformer {
    String name;        // the persistent storage,
                        // a permanent part of each NameDropper

    NameDropper(String whatMyNameShouldBe) {  // the creation rule
        this.name = whatMyNameShouldBe;
    }

    String transform (String whatToSay) {     // the transform rule
        return this.name + " says " + whatToSay;
    }
}
```

Now, when we invoke NameDropper's construction method, we give it an argument:

```
new NameDropper("Pat")
```

for example.

We have actually seen — or at least alluded to — a similar situation earlier. When discussing the other entities that together constitute a Transformer, we said that the input Connection Acceptor's job was to stick the input Connection it receives somewhere where the rest of the Transformer community can use it. Like NameDropper, the generic StringTransformer accomplishes this using a *field*.

Fields, methods, and constructors are the building blocks of Java objects. We will see each of these things in action in the next several chapters. In Chapter 7, *Building New Things: Classes and Objects*, we will go through each of these items in greater detail. For now, it is enough to have a general sense of how things fit together.

A.3.6 Generality of the approach

In writing this code, we have relied on the existence of a generic StringTransformer class. In that class, we include rules for how to accept an input Connection (using a field to store it away), how to accept an output Connection, and how to create an individual StringTransformer, including creating its own instruction−follower to explicitly invoke the *transform* method over and over again on each String read from the stored input Connection. The ways in which this StringTransformer class is put together are much like the ways in which the examples here are constructed, but the StringTransformer class is about four times the size of the classes described above. The complete code for StringTransformer is included in the on−line supplement to this book.

The Transformers that we have written here each obey the same general rules and interfaces. Each defines a *transform* method that takes a String and returns a String. The apparent uniformity among StringTransformers makes it possible for the Connection

mechanism that we outlined in the previous section to work with each of them. The differences among StringTransformer behaviors are hidden inside the *transform* method that each of them implements. In the course of this book, we will see many different cases in which **hiding behavior behind a common interface makes a system more general and more powerful. Good design specifications are crucial; they amount to deciding in advance how entities will interact.**

Chapter Summary

In this Interlude, you have been exposed to many of the most basic pieces of Java programming.

- **None of these has been presented in sufficient detail to achieve mastery of it.**
- **Each of these topics will be revisited, most in the next part of the book.**

But the example described above gives a context within which to place the detail that occupies the next several chapters.

Here is what's ahead:

- In the next chapter, which begins Part 2 of this book, we will explore the role of ***types*** in Java systems and the relationship between ***types*** and ***names***.
- The next chapter in Part 2 (Chapter 4) looks at ***interfaces***, the contracts that one type of object makes with another.
- In Chapter 5, we turn to ***expressions*** — such as method invocation, field access, instance construction, and even String concatenation — and learn how evaluating an expression produces a value of a specified type.
- Expressions are combined to make the topic of Chapter 6: ***statements***, the step−by−step instructions of Java code that produce behavior and flow of control.
- Chapters 7 and 8 discuss ***classes***, which allow us to implement behavior and to encapsulate both instructions and local state — such as the NameDropper's name — into individual ***objects***.
- And ***self−animating objects***, discussed in Chapter 9, contain their own instruction−followers that execute sequences of instructions over and over, communicating with other objects and interacting to provide desired behavior on an ongoing basis.

Exercises

See the text for things marked **Question:**. Also:

1. Implement LowerCaser.
2. Implement SentenceCaser: the first letter is capitalized, while the rest are not.
3. Implement the simplest version of Pig Latin.
4. An improved Pig Latin would leave the first letter in place if it were a vowel, and add −*way* instead. This requires understanding basic conditionals and flow of control: see Chapter 6, *Statements and Rules*.
5. Ubby Dubby is pretty hard. You may want to look carefully at Chapter 12, *Dealing with Difference: Dispatch*, as well as earlier chapters.
6. Combiners and Repeaters involve extending StringTransformer in other ways, overriding *acceptInputConnection* or *acceptOutputConnection*.
7. Really challenging problem: extract words, one word at a time, only reading an input when all words have been used up.

Part 2

Entities and Interactions

Chapter 3

Things, Types, and Names

Chapter Overview

- What kinds of Things can computers talk about?
- How do I figure out what they can do (or how they interact)?
- How can I keep track of Things I know about?

This chapter introduces some of the conceptual structure necessary to understand Java programs. It begins by considering what kinds of things a program can manipulate. Some things are very simple—like numbers—and others are much more complex—like radio buttons. Simple (***primitive***) things can't do anything by themselves, but in later chapters you'll learn how to do things with them. Many complex things can actually act, either by themselves (e.g. a clock that ticks off each second) or when you ask them to (e.g. a radio that can play a song on request). These complex things are called ***objects***.

The remainder of this chapter introduces two important concepts for understanding and manipulating things in Java: ***typing*** and ***naming***.

Types are ways of looking at things. A type specifies what a thing can do (or what you can do with a thing). Types are like contracts that tell you what kinds of interactions you can have with things. Sometimes, the same thing can be viewed in different ways, i.e., as having multiple types. For example, a person can be viewed as a police officer or as a mother, depending on the context. (When making an arrest, she is acting as a police officer; when you ask her for a second helping of dessert, you are treating her as a mother.) A thing's type describes the way in which you are regarding that thing. It does not necessarily give the complete picture of the thing.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Names are ways of referring to things that already exist. A name doesn't bring a thing into existence, but it is a useful way to get hold of a thing you've seen before. Every name has an associated type, which tells you what sorts of things the name can refer to. It also tells you what you can expect of the thing that that name refers to. In other words, the type describes how you can interact with the thing that the name names. There are actually two different kinds of names in Java: ***primitive*** (***dial***) names and ***reference*** (***label***) names.

Sidebars in this chapter cover the details of legal Java names, Java primitive types, and other syntactic and language−reference details.

Objectives of this Chapter

1. To recognize Java types.
2. To distinguish Java primitive from object types.
3. To be able to declare and define names.
4. To understand that a declaration permanently associates a type with a name.
5. To recognize that each dial name contains exactly one value at any time.
6. To understand how a label name can have a referent or have no referent (i.e., be `null`).
7. To be able to tell when the values associated with two names are equal.

3.1 Things in Programs

In the first part of this book, we have explored what computational systems are and how they are created. In this part, we will shift our focus to the things out of which such systems are created. The previous part took a top−down approach to system design. In this part, we will learn the basic building blocks that will enable us also to approach the problem from the bottom up.

In building a computer program, we will make use of many things. Some of these things are relatively simple and, when we need them, we can just write them down. For example, `4` and `-6.3` are things we might want in our programs, and we can put them there just by writing them down in the appropriate place. But other things — like the library's record for *Moby Dick* or the current location of the cursor — are harder to write down explicitly, in the way that we were able to write `4` or `-6.3` . For these kinds of

things, we will need a way to refer to them without explicitly writing them down each time. We do this using a *name*. A ***name*** is a way to refer to something in your program. At any given time, a name refers to at most one thing. What the name refers to may change over time, though, so *the current location of the cursor* may be `(26, 155)` now and `(101, 32)` later. This is true in real life, too: *the student whose birthday comes next* changes every time someone celebrates becoming a year older.

In this chapter, we explore how names refer to things in Java. We also explore some specific kinds of things that can be included in a Java program. Although there are details in this chapter that are particular to Java, many of the principles we identify work in almost any computer programming language. These include:

- Names can be used to refer to things, allowing you to hold on to a thing even if you can't spell it out entirely.

- Some things can be written directly into your program. These are called ***literals***.

In addition, in Java and in many other languages, each name can be used only to refer to things of a particular type. A language in which a name is restricted to refer to things of a particular type is called a ***strongly typed language***. In a strongly typed language, live Java,

- A name must be ***declared***, meaning that you must explicitly say in the program what type of thing this name can refer to.

3.2 Most Java Things are Objects

Different programming languages allow you to talk more or less easily about different kinds of things. Some languages — like Basic — are very restricted in the kinds of things that a program can talk about explicitly. Other languages — like Java — allow you to create and talk about almost any kind of thing that you can imagine. Java is one of a family of languages called ***object oriented programming languages***. In Java, as in most object oriented programming languages, most things — and all of the things that you will create — are called ***objects***. Almost anything that you can describe in a programming language can be an object, so, in Java, ***object*** is a pretty general word for "programming language thing." Not all programming languages allow you to talk about objects, though, and — towards the end of this chapter — we will learn about a few Java things that are not objects.

An object can be just about anything that you might want to represent in a computer program. Some example objects include *the radio button that the user just clicked*, *the window in which your program is displaying its output*, and *the URL of your home page*.

Objects are frequently complex things with internal state. For example, the window may have a "close" box or a background color; the URL has a host computer name. Many

objects also know how to do things. The radio button may be able to tell you whether it has been selected, or the window may know how to close. An ***object*** is a programming language thing that can have complex internal state.

Some objects can even act on their own without being asked to do anything; they are "born" or created with the ability to act autonomously. For example, an Animator may paint a series of pictures rapidly on a screen, so that it looks to a human observer like the picture is actually moving. The animator may do this independently, without being asked to change the picture every 1/30th of a second. Similarly, an alarm clock may keep track of the time and start ringing when a preset time arises.

We will begin by using objects that are provided to us, because it is important to learn how to interact with an object. Many of the objects that you will use throughout your programming career will be designed by other people for you to use. Later — in Chapter 7, *Building New Things: Classes and Objects* — we will learn how to build objects of our own.

Almost everything that you do in Java uses objects, and you will hear much more about them throughout this book. This chapter concentrates on how you identify the things (including objects) in a program and how names can be used to refer to them. The next chapter looks at ***interfaces***, the contracts that one type of object makes with another. In Chapter 5, *Expressions: Doing Things with Things*, we will see in more detail how to use things (including objects) to produce other things. Chapter 6, *Statements and Rules*, concentrates on combining these pieces into a full−blown recipe, a single list of instructions that can be followed to accomplish a particular job. The three chapters following (Chapters 7 through 9) look at objects in more detail, describing how to create and use the objects that are manipulated by these instructions, and how these instructions themselves can be combined to form objects and entities that interact in a community.

3.2.1 Doing Things with Objects

There are several ways to talk about an object. One is to call it by its name. For this to work, the object must have a name, but fortunately, many objects do. For example, the object that causes text to appear on your screen can be addressed — in this textbook and its associated packages — using the name ***Console*** . Information about what you can do with *Console* is contained in the sidebar below.

A name like this gives you direct access to the object it names. For example, I might have a robot that I call *robbie*; then I can ask *robbie* to move using its name. I can also tell you about *robbie* using this name. Below, you will see how to give names to things, and — using that technique — I could even give *robbie* the nickname *fred* by giving the robot a second name. A name like *robbie* or *Console* names the object, and is like a label that allows you to access the object directly.

There are also other ways to refer to an object without using a name for it. You can do this, for example, by using a related object to help you. Imagine that *mobyDick* names a book. Then

```
mobyDick.author()
```

might refer to the author of *Moby Dick*. The dot (`.`) and parentheses () are special ***syntax*** (that is, notation) that indicate that we're asking the thing called *mobyDick* to tell us its author. We will explore this syntax further in the next few chapters, but for now you can regard it as a special incantation to use when you want to ask a thing for something that it knows how to give you. (In the next chapter, we'll explore how you can find out what kinds of actions a particular kind of thing can perform.)

Of course, we might be able to refer to the author of *Moby Dick* not only indirectly by **`mobyDick.author()`** but also by using a name designated for this purpose, like *hermanMelville*.

The name *mobyDick* might always refer to the same thing, as would **`mobyDick.author()`**. But not all names have constant values. Imagine that *numberGuesser* is the name of an object that knows how to guess numbers. The name *numberGuesser* might even always refer to the same guesser. But you'd hope that an expression like **`numberGuesser.guess()`** would refer to different values from one guess to another, or the thing named *numberGuesser* wouldn't be much of a number guesser! And if I come up with a better strategy for guessing numbers, I could make the name *numberGuesser* refer to an entirely new object, too. We'll see more about giving things names and changing the values referred to by names below.

One particularly useful kind of object is called a *String*. A ***String*** is a sequence of characters. To describe a specific String in Java — for example, the message that your computer prints to the screen when you boot it up — you can write it out surrounded by double quotation marks: `"Hi, how are you?"` or `"#^$%&&*%^$"` or even `"2 + 2"`. Note that the quotation marks are not actually part of the String; they're just there to make it clear where the String begins and ends.

Your computer doesn't *understand* the String, it just remembers it. For example, the computer doesn't know of any particular relationship between the last example (`"2 + 2"`) and the number `4` — or the String `"4"`.

Strings are useful, for example, to communicate with a program's user. Error messages, user input (i.e., what you type to a running Java program), titles and captions are all examples of Strings.

Strings can do some interesting things. For example, if *myName* is `"Rigoberto Manchu"`, then **`myName.toLowerCase()`** is `"rigoberto manchu"` and **`myName.length()`** is 16. Strings are part of the Java language, so they are available to

every Java program.

Another particularly useful object is *Console*, which is part of the ***cs101 libraries***.

[Footnote: This means that *Console* is available only to Java programs that use the cs101 libraries (so named because they were developed as part of MIT's cs101 course). See the sidebar on *Console* for how to obtain and use those libraries. Later in this book, we'll learn how to replicate the function of *Console* using only things built in to Java, in case you don't always want to use the cs101 libraries.]

Console is an object that can print a String to the ***Java console***, a standard place where someone running a Java program can look for information. *Console* can also *readln* a String that the user types to the Java console.

Console

Console is a special object that knows how to communicate with the user in some very basic ways. If your program says

```
Console.println("Hello there!");
```

then the String `"Hello there!"` will appear on the Java console. (Remember, the double quotation marks aren't part of the String; they're just used to indicate where it starts and ends.) The statement

```
Console.print("Hi");
```

is similar, except that **`Console.print`** doesn't end the line of output, while **`Console.println`** does. This means that

```
Console.print("A ");
Console.print("is for apple.");
```

would produce the output

```
A is for apple.
```

while

```
Console.println("A ");
Console.println("is for apple.");
```

would produce

```
A
is for apple.
```

You can of course combine *print*s and *println*s arbitrarily. Printing a String containing a newline character escape (`\n`) causes the line to end as well.

You can also use Strings that are associated with names or any other Strings you may have access to, not just String literals.

To use *Console*, you must install the cs101 libraries in your Java system and import the ***cs101.util.Console package***. Your instructor may have set this up for you or can show you how to do so yourself. However, in all of the above examples, you could replace *Console* by the name *System.out*, which is part of the standard Java libraries (and hence needs no special action on your part to use it).

Console's most important virtue, besides being a good example of an object, is that you can use it to capture Strings that the user types on the Java console. In particular, the expression

```
Console.readln()
```

returns a String, specifically the String typed by the user (and ending with a *return* or *enter* character) on the Java console.

3.3 Naming Things

With all of these things floating around in our program, it is pretty easy to see that we'll need some ways to keep track of them. The simplest way to keep track of things is to give them names. This is called ***assigning*** a value to a name. Giving something a name is sort of like sticking a label on the thing. We sometimes say that the name is ***bound*** to that value.

Java Naming Syntax and Conventions

Java identifiers can contain any alphanumeric characters as well as the symbols $ (dollar sign) and _ (underscore). The first character in a Java identifier cannot be a number. So *luckyDuck* is a legitimate Java identifier, as is *_Alice_In_Wonderland_*, but *24T* is not.

Certain names in Java are ***reserved words***. This means that they have special meanings and cannot be used as names — i.e., to refer to things, other than any built−in meaning they may have — in Java. Reserved words are sometimes also called ***keywords***. These are:

```
abstract   default    if           private      throw
boolean    do         implements   protected    throws
break      double     import       public       transient
byte       else       instanceof   return       try
case       extends    int          short        void
catch      final      interface    static       volatile
char       finally    long         super        while
class      float      native       switch
const      for        new          synchronized
continue   goto       package      this
```

Java is ***case−sensitive***. This means that *double* and *Double* are two different words in Java. However, you can insert any amount of ***white space*** — spaces, tabs, line breaks, etc. — between two separate pieces of Java — or leave no space at all, provided that you don't run words together. You can't stick white space into the middle of a piece of Java — a name or number, for example — though.

Punctuation matters in Java. Pay careful attention to its use. Note, however, that white space — spaces, tabs, line breaks, etc. — do *not* matter in Java. Use white space to make your code more legible and easier to understand. You will discover that there are certain conventions to the use of white space — such as lining up the names in a column, as we did above — although these tend to vary from one programmer to the next.

A few names come already bound. *Console* is one example (provided that you are using the cs101 libraries). Throughout this book, you will discover a few other pre−bound names. But for other things, you will want to create your own names so that you can continue to refer back to them.

To actually ***assign*** a value to a name — to create a binding between that name and that value — Java uses the ***syntax*** (that is, notation):

Assignment
name = value

Recall the *numberGuesser*, above. We can ask the *numberGuesser* to guess a number, but if we don't do anything with that guessed number, we will have no way to refer back to it. It is as if soon after getting the guessed number from *numberGuesser*, it is dropped onto the floor with all the other things that our program has created or used. Unless we have somehow marked what *numberGuesser* gives us, we cannot get it back later.

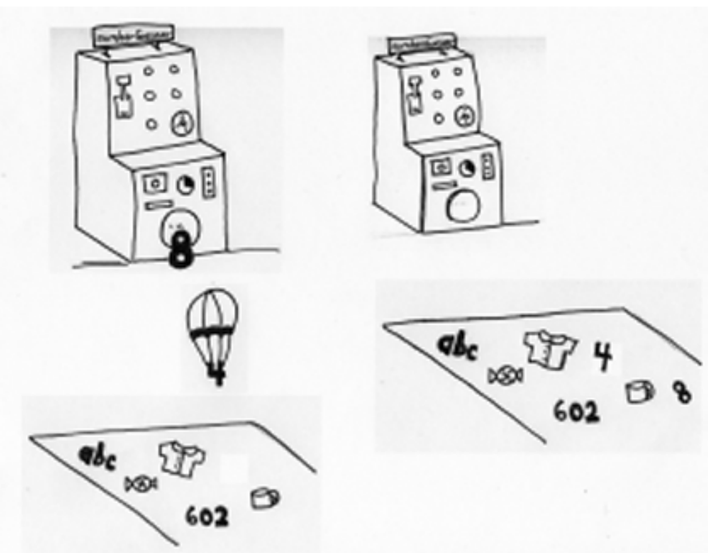


Figure 3.1. If we don't do something with our things,
they get mixed up with the other things we used.

You can solve this problem by putting a label on the first guess when you get it. Then, if you want it back later, you can ask for it by name: the name on the label. That is, we can remember the guess our *numberGuesser* made by giving it a name:

```
myFavoriteNumber = numberGuesser.guess()
```

This associates the value guessed with the name *myFavoriteNumber*.

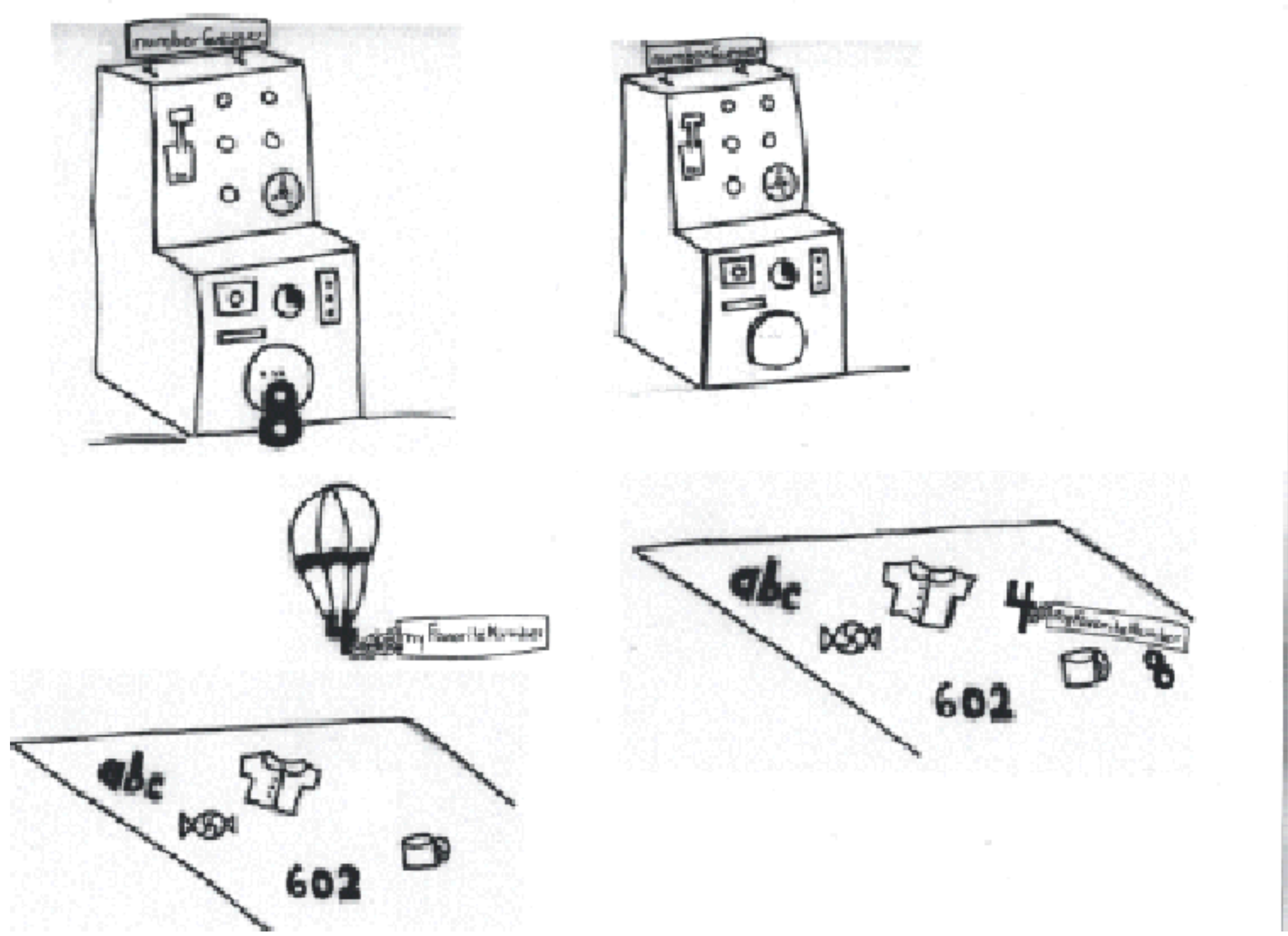


Figure 3.2. By giving a thing a name, we can refer to it later.

If you want to continue to have access to something later, you'll need to give it a name.

Once a particular name refers to a particular thing — say *greeting* has the value `"Hi, how are you?"` — then we can use the name wherever we would use its value, with the same effect. The name becomes a stand−in for the thing it refers to. In Chapter 5, *Expressions: Doing Things with Things*, we will see that a name is a simple kind of ***expression***.

A name, like a label, can be affixed to only one thing at a time. In other words, only one value may associated with a name at any given time. One thing can be referred to by any number of names at once (including, potentially, no names at all). The same person can be “the person holding my right hand,” “my very best friend,” and “Chris Smith.” But only one person is “the person holding my right hand.”

[Footnote: Barring weird interpersonal pileups, of course.]

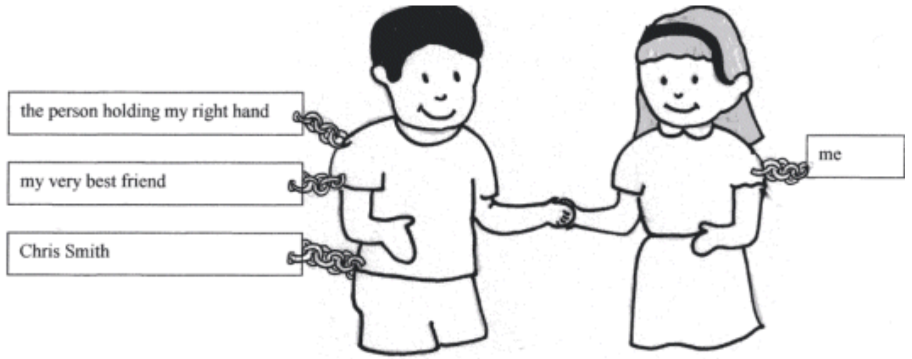


Figure 3.3. A name refers to only one thing at a time.
But several different names can refer to the same thing.

But before we can assign a value to a name, we need to know whether the name is allowed to label values of that type.

3.4 Types

Up to now, we've been pretty casual about our things. Java, however, is a strongly typed language, meaning that it is not at all casual about what kind of thing something is. Each Java thing comes into the world with a ***type***, i.e., an indication of what kind of thing it is. Java names, too, are created with types, and a Java name can only be used to label things of the appropriate type. Before we can use a name — as *myFavoriteNumber*, above — we have to ***declare*** it to be of a particular type. Declaring a name means stating that that particular name is to be used for labeling things of some particular type.

3.4.1 What a Type Is

We have been using the idea of *types* all along, but in this section we make that idea concrete and specific.

The ***type*** of a thing is an indication of what kind of thing it is. In particular, the ***type*** of a thing:

- Tells you what kind of behavior you can expect from the thing.
- Tells you what properties the thing has.
- Provides a means for knowing what you can do with the thing.
- Tells your computer something about how it should represent and manipulate the thing internally.

In English, we can refer informally to the kind of thing that something is, like "a number" or "a description of a music CD." In a programming language, we need to be more precise. For example, one particular type of number in Java is called a ***short*** and can be described more precisely as "a whole number between `-32,768` and `32,767` represented by 16 bits." (See the sidebar on *Java Primitive Types* for details.)

In this chapter, we'll be using types that someone else has defined. In the next several chapters, we'll be learning how to figure out what a certain type of thing is like and what you can do with it. In Chapter 7, *Building New Things: Classes and Objects*, we will at last learn how to build our own types of things.

3.4.1.1 Two Kinds of Types: Primitive Types and Object Types

Some types are very specific and have very limited functionality. Such types are called ***primitive types***. The `short` type mentioned above is a primitive type: a `short` cannot do anything on its own, although you can use `shorts` for arithmetic in the usual way. There are exactly eight primitive types in Java. Later in this chapter (in Section 3.6), we will see the details of the primitive types. In Chapter 5, *Expressions: Doing Things with Things*, we'll see how to manipulate primitive-type things.

Most of what goes on in Java, though, concerns not primitive-type things, but rather ***object-type*** things. As we've seen, an object can be just about anything that you might want to represent in a computer program. Some example objects include *the radio button that the user just clicked*, *the window in which your program is displaying its output*, and *the URL of your home page*. Every object has an ***object type*** as its type.

Every type in Java is either a primitive type (as described in Section 3.6) or an object type (as described in the next subsection).

3.4.1.2 Object Types

Java's standard libraries provide thousands of predefined ***object types***, such as the String type that we saw earlier in this chapter and the Button and JButton types that describe clickable (GUI) objects that might appear in a window on your computer screen. If you are using the cs101 libraries, you'll also have access to object types such as AnimateObject — something that can move by itself — and DefaultFrame — an easy kind of window to use. And, in the rest of this book, you will be learning to define your own object types to do what you want. These object types — whether a part of the Java language or of your own definition — describe kinds of objects.

Some examples of object types might include KlingonStarship (if you're building a space battle adventure game), IllustratedBook (if you're building an electronic library system), or PigLatinTranslator (if you're building a networked chat program). Each of these object types may describe many different individual objects — the three KlingonStarships visible on your screen, the five hundred and seven IllustratedBooks in the children's

library, or the particular PigLatinTranslator that your particular chat program is using. These individual objects are sometimes called ***instances*** of their types. For example, the KlingonStarship that you just destroyed is a different KlingonStarship instance from the one that is getting ready to fire its phasers at you. We'll explore this idea in greater detail in Chapter 7, *Building New Things: Classes and Objects*.

Each kind of object — each object type — determines what services or actions individual objects of that type can do. For example, Windows can close; Dictionaries can do lookups; KlingonStarships can fly around the screen. Further, each individual object of that type can perform these actions. For example, if *myWindow* and *yourWindow* are two different window−type objects, *myWindow* can close, and so can *yourWindow*. But if *myWindow* closes, that doesn't in general affect *yourWindow*.

Each individual object comes ready−made with all the properties and behavior that that objects' object type specifies. An IllustratedBook has an author and an illustrator, for example. A PigLatinTranslator may be able to translate a word that we supply it into Pig Latin. We ask objects to do things (including telling us about themselves) using specific services that these objects provide. Often, these services are accessed by giving the name of the object we're asking followed by a dot (or period), followed by the request we're making of the object. So if *theLittlePrince* is the name of an IllustratedBook,

```
theLittlePrince.getAuthor()
```

would be a request for the name of the author of the book: `"Maurice de Saint Exupery"`. Similarly, if *myTranslator* is a PigLatinTranslator,

```
myTranslator.transform("Hello")
```

might be a request to *myTranslator* to produce the Pig−Latin−ified version of `"Hello"`, which is `"ello-Hay"`. These requests are the most basic form of interaction among the entities in our community.

3.4.1.3 What a Type Is: Summary

In summary, the ***type*** of a thing is an indication of what kind of thing it is. In particular, the ***type*** of a thing:

- Tells you what kind of behavior you can expect from the thing.
- Tells you what properties the thing has.
- Provides a means for knowing what you can do with the thing.
- Tells your computer something about how it should represent and manipulate the thing internally.

Some things (e.g., shorts) are ***primitive−type things***, but most things in Java are ***object−type things***. String and Button are examples of predefined object types that are part of the standard Java libraries, while KlingonStarship might be an example of an object type that you might define yourself. Only object−type things (which are called *objects*) have behaviors or properties of their own; primitive−type things can be used but cannot do anything by themselves. We can use the dot−notation, as shown in the above examples, to ask objects to do things for us.

3.4.2 Types of Objects

Every Java thing comes into the world with a type. As we will see in Chapter 7, *Building New Things: Classes and Objects*, you use the ***new*** keyword to construct a new object. For example, the expression

```
new Cat()
```

constructs a new object whose type is Cat.

The type of an object never changes. So, for example, the object created by the above expression is a Cat and always will be a Cat. However, as we've seen, several names can refer to the same object. These names each have their own type, as described in the next section, and can be different types. For example, we might refer to the above Cat at one time by a name that is a Pet and at another time by a name that is an Animal, depending on what we need the Cat to do. Don't worry if this seems confusing; we'll discuss it in much greater detail in Chapter 8, *Designing with Objects*.

3.4.3 Types of Names

As we have seen, each Java object comes into the world with a type, i.e., an indication of what kind of thing it is. Every Java name has a type, too, and a Java name can be used only to label things of the appropriate type. The type of a name is associated with the name when the name is ***declared***, as described shortly. **The type associated with a particular name never changes.**

3.4.3.1 Declarations and the *Type−of−Thing Name−of−Thing* Rule

Before we can use a name — *myFavoriteNumber*, for example — we have to ***declare*** it to be of a particular type. Declaring a name means stating that that particular name is to be used for labeling things of some particular type.

Names are declared using the ***Type−of−thing Name−of−thing*** rule:

```
int myFavoriteNumber;
Cat marigold;
```

The second word on each line is a name that is being declared. The first word on each line is the type that the name is being declared to have. In the first line of the example above, *myFavoriteNumber* is being declared to have type ***int***. This is the Java primitive type that we usually use for whole numbers (integers).

So the first declaration here creates a name, *myFavoriteNumber*, suitable for naming integers (or, more precisely `int`s). The second line creates the name *marigold*, suitable for naming objects of type Cat.

As the example shows, the primitive types begin with a lower−case letter, while object types traditionally begin with an upper−case letter. Also, each declaration ends with the ***semicolon*** (***;***) that concludes each ***statement*** in Java. More on semicolons and statements in Chapter 6, *Statements and Rules*.

A name has a certain lifetime, sometimes called its ***scope***. Within that scope — over its lifetime — the name may be bound to many different values, though it can only be bound to one value at a time. For example, *myFavoriteNumber* may initially be 4, but later change to be 13. The association between a name and a type persists for the lifetime of the name, however. Thus, *myFavoriteNumber* can only name an `int`, not a String or a `short` or a Cat.

3.4.3.2 Definition = Declaration + Assignment

Declaring a name begins its useful lifetime. At that time, nothing else necessarily needs to happen — and frequently, it doesn't. But sometimes it is useful to associate the name with a value at the time that it is declared. This combination of a declaration and an assignment is called a ***definition***.

- A declaration tells you what type is associated with a name.
- An assignment sets the value of a name, thereby telling you what value that name is bound to.
- A ***definition*** combines the “what kind of thing it can name” and “what value it has” statement types.

For example:

```
boolean isHappy = true;
double  degreesCelsius = −273.18;
Thread  spirit = new Thread(this);
Cat     myPet = marigold;
```

The first and second of these make use of ***boolean*** and ***double*** constants, respectively, to assign values to the names *isHappy* and *degreesCelsius*. Both `boolean` and `double` are primitive types described in the sidebar on *Java Primitive Types* later in this chapter.

The Thread definition creates a new Thread using the `new` keyword (much more on this later) and associates the name *spirit* with that newly created Thread.

The final definition makes the name *myPet* refer to the same Cat currently named by *marigold*. This is an example of the name *marigold* standing in for the actual Cat , that is, the name being used in place of the thing it refers to. After the assignment completes, *myPet* is bound to the actual Cat, not to the name. If the name *marigold* later refers to some other Cat — say both Cats undergo name changes — the name *myPet* will still refer to the Cat originally known as *marigold*.

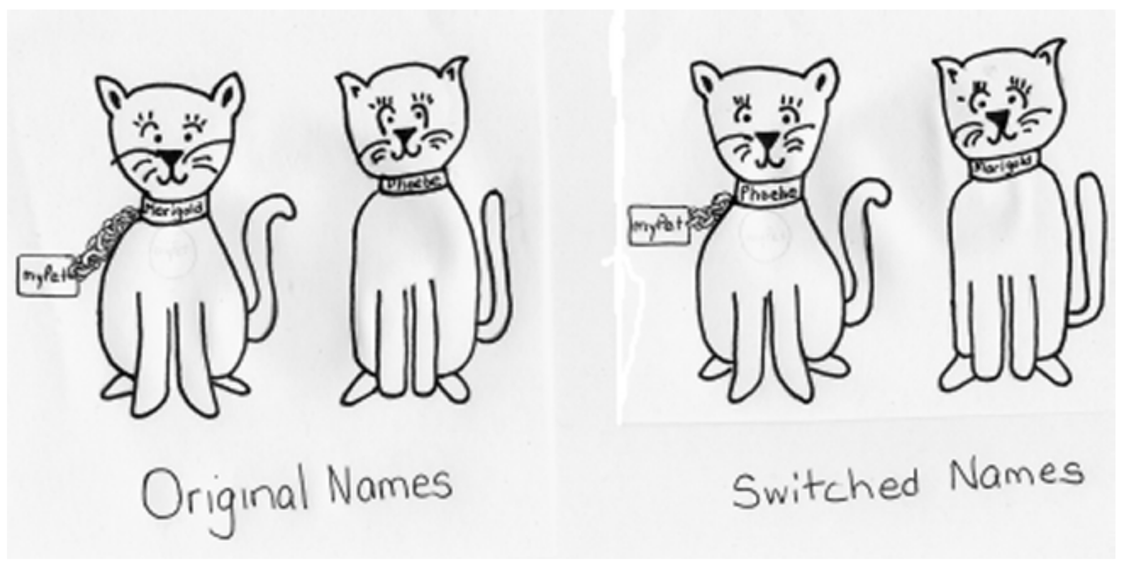


Figure 3.4. *myPet* labels the Cat, not the name.
Changing the name of the Cat does not affect *myPet*.
So the name *myPet* still refers to the curved−tail cat, even if the cats switch names.

3.5 Names for Objects: Label Names

We have seen that there are two kinds of things in Java: object−type things (which are called objects) and primitive−type things. Likewise, there are two kinds of names in Java: names that are declared to be some object−type — which we will call ***label names*** — and names that are declared to be some primitive−type — which we will call ***dial names***. This section examines the former; Section 3.6 examines the latter.

When a name is declared to be some object−type, a new *label* suitable for affixing on things with that type is created. We will call such names ***label names*** or simply ***labels***. For example, a building name might be a cornerstone label, a person's name might go on a badge, and a dog's name might belong on a collar. You can't label a person with a cornerstone or pin a badge on a dog, at least not without raising an error. Unlike cornerstones or dog tags, though, labeling a Java object doesn't actually change that object. It just gives you a convenient way to identify (or grab hold of) the object.

In Java terms, if we declare

```
RadioButton myButton;
```

this creates a label, *myButton*, that *can be* stuck onto things of type RadioButton. It is *not* currently so stuck, though. At the moment, *myButton* is a label that isn't stuck to anything. (Cornerstones and badges and dog tags don't come with buildings and people and dogs attached, either. Having a label is different from having something to label with it.) **Labels don't (necessarily) come into the world attached to anything.** The value of a label not currently stuck onto anything is the special non−value ***null***. (That is, `null` doesn't point, or refer, to anything.) So the declaration above is (in most cases) the same as defining

```
RadioButton myButton = null;
```

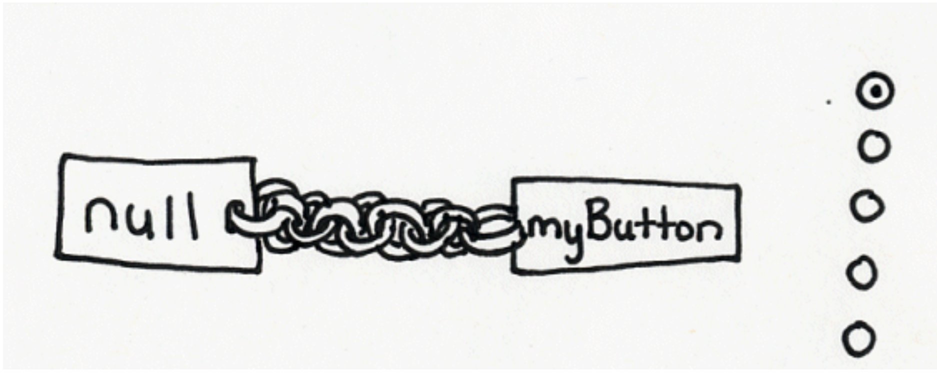


Figure 3.5. A label name (*myButton*) that's not yet stuck on anything.
Its type is RadioButton, so it *could* become stuck onto any of the RadioButtons shown.
But currently it is *not* stuck on any of them, so its current value is `null`.

Of course, we can attach a label to something, though we need to have that something first. We'll return to the question of where things come from in a few chapters. For the moment, let's suppose that we have a particular object with type RadioButton, and we stick the *myButton* label onto it. (Now *myButton*'s value is no longer `null`.)

After we give *myButton* a value — stick it onto a particular RadioButton — we can refer to it. For example, we can check to see whether it's pressed:

```
myButton.isSelected()
```

(This is an expression that returns a boolean value; see the discussion of expressions in Chapter 5, *Expressions: Doing Things with Things*.)

If we now declare

```
RadioButton yourButton = myButton;
```

a new label is created. This new label is attached to the *same* object currently labeled by *myButton*. **Assignments of label names do not create new (copies of) objects.** In this case, we have two labels stuck onto exactly the same object, and we say that the names *myButton* and *yourButton* ***share a reference***. This just like saying that “the morning star” and “the evening star” both refer to the same heavenly body.

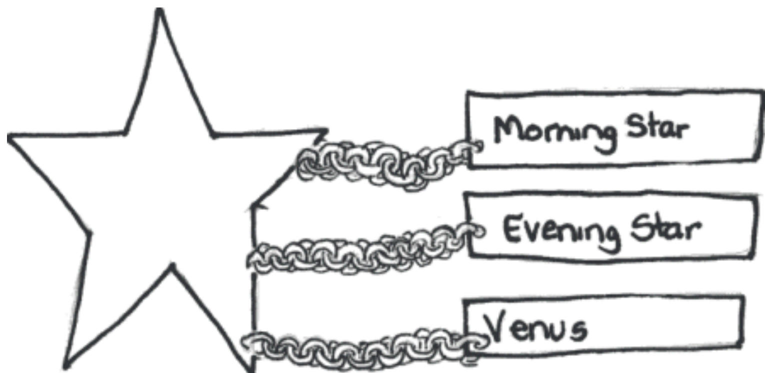


Figure 3.6. Multiple labels can refer to the same object.

Because *myButton* and *yourButton* are two names of the same object, we know that

```
myButton.isSelected()
```

and

```
yourButton.isSelected()
```

will be the same: either the button that both names label is pressed, or it isn't. But we can separate the two labels — say

```
myButton = someOtherButton;
```

— and now the values of **`myButton.isSelected()`** and **`yourButton.isSelected()`** might differ (unless, of course, *someOtherButton* referred to the same thing as *yourButton*). Note that moving the *myButton* label to a new object doesn't have any effect on the *yourButton* label.

Note also that the labeled object is not in any way aware of the label. The actual RadioButton doesn't know whether it has one label attached to it, or many, or none. A label provides access to the object it is labeling, but not the other way around.

3.6 Primitive Types, Literals and Dial Names

Objects are extremely useful, and every Java program that you use will make use of objects. But there are a few other kinds of things that do not have the same complex internal structure or behavior that objects have. These other kinds of things are called

primitive−type things. This section describes such things and how to refer to them, by using ***literals*** (Section 3.6.1) and by using ***dial names*** (Section 3.6.3).

3.6.1 Literals

Java, like many programming languages, has some built−in facilities for handling and manipulating simple kinds of information. For example, Java knows about numbers. If you type `6` in an appropriate place in a Java program, the computer will "understand" that you are referring to an integer greater than 5 and less than 7. The expression `6` is a Java ***literal***: an expression whose value is directly "understood" by the computer. In addition to integers, Java recognizes literals that approximate real numbers expressed in decimal notation as well as single textual characters.

This means that all of the following are legitimate things to say in Java:

- `6`
- `42`
- `3.5`
- `−3598.43101`

Details of Java numeric literals — and of all of the other literals discussed here — are covered in the sidebar on *Java Primitive Types*. As we will see in Chapter 5, *Expressions: Doing Things with Things*, you can perform all of the usual arithmetic operations with Java's numbers.

[Footnote: Be warned, though, that non−integral values, like 1/3 and 1.234567890123456789, are in general represented only approximately.]

Java can also manipulate letters and other ***characters***. When you type them into Java, you have to surround each character with a pair of single quotation marks: `'a'`, `'x'`, or `'%'`, for example. This enables Java to tell the difference between `6` (the integer between 5 and 7) and `'6'` (the character 6, which on my keyboard is a lower case '^'). The first is something that you can add or subtract. The second is not.

It turns out that it's also useful for many programs to be able to manipulate conditions, too, so Java has one last kind of primitive value. For example, if we are making sandwiches, it might be important to represent whether we've run out of bread. We can talk about what to do when the bread basket is empty:

> ***if*** the bread basket is empty, buy some more bread ...

Conditions like this — bread−basket emptiness — are either true or false. We call this kind of thing a ***boolean*** value. Booleans are almost always used in ***conditional*** — or ***test*** — statements to determine ***flow of control***, i.e., what should this piece of the program do next? Java recognizes ***true*** and ***false*** as boolean literals: if you type one of them in an appropriate place in your program, Java will treat it as the corresponding truth value.

As we have just seen, primitive−type things can be referenced by literals. In addition, Strings can be referenced by literals, by using the double−quotation mark syntax:

```
"I loves you porgy"
```

So all three of the following are literals:

```
"5"     '5'     5
```

but the first is an object−type thing (a String), while the latter two are primitive−type things (a `char` and an `int`, respectively).

There are lots of rules about how these different things work and how they are used. For many of the detailed rules about the primitive things that we have just covered, see the sidebar on *Java Primitive Types*.

3.6.2 Primitive Types

A type is Java's way of indicating what kind of thing something is and what it can do. Like objects, Java primitive things have types. But unlike objects, you cannot create any new Java primitive types. Java has exactly eight ***primitive types***. These types are built into the Java language, so they are always available to you. Four of these types correspond to integers. Two of the types correspond to decimal numbers. One of types is for single characters. The eighth type is for true−or−false values, or booleans. These are all and exactly the primitive types permitted in Java. For details on these types, including their names and their properties, see the sidebar on *Java Primitive Types*.

Java objects may have complex internal state or the ability to perform interesting behaviors. Java primitive−type things do not have any internal state, nor can they do anything by themselves. They cannot ring like an alarm clock, close like a window, or be selected like a radio button. They cannot even add themselves or display themselves on a screen. Only objects can be asked to do things for themselves. In Chapter 5, *Expressions: Doing Things with Things*, we will learn how we can use primitive things to accomplish useful tasks. But, unlike object−type things (objects), primitive−type things cannot accomplish anything by themselves.

Java Primitive Types

Each Java primitive type has its own built-in name. For example, ***int*** is a name for a type-of-thing corresponding to an integer value. There are actually four Java names for integers, depending on how much space the computer uses to store them. An `int` uses 32 ***bits***, or **b**inary dig**its**. An `int` can represent a number between `-2,147,483,648` and `2,147,483,647`, that is, from -2^{31} to $2^{31} - 1$, which is big enough for most purposes. (You can't type commas in literal `int`s, however.) An integral number (i.e., a number without a decimal point) appearing literally in a Java program will be interpreted as an `int`.

If you need a larger range of numbers, you can use the Java type ***long***, which can hold values between -2^{63} and $2^{63} - 1$. You can't just type in a value like `80951151051778`, though. Literals intended to be interpreted as `long` must end with the character ***L*** (or ***l***):

```
80951151051778L
```

There are also two smaller integer types: the 16-bit ***short*** and the 8-bit ***byte*** . There are no `short` or `byte` literals. For most purposes, the `int` is probably the Java integral type of choice. **The limited range of all the integral types means that calculations using such numbers can *overflow* and give wrong results. The programmer must be aware of such limits.**

Approximations to real numbers are represented internally using a notation called ***floating point*** notation (so named because the decimal point can “float” as in scientific notation). There are two primitive types for representing floating point numbers, again corresponding to the amount of space that the computer uses to store them. One is ***float***, short for *floating point*; the other is ***double***, for ***double precision*** floating point. Both are only approximations to real numbers, and `double` is a better approximation than `float`. Neither is precise enough for certain scientific calculations.

A `float` is 32 bits. The biggest `float` is about 3.4×10^{38}; the smallest is about -3.4×10^{38}; The `float` type can represent numbers to an accuracy of about 8 significant decimal digits.

A `double` is 64 bits. The biggest `double` is about 1.8×10^{308}; the smallest is about -1.8×10^{308}. The `double` type can represent numbers to an accuracy of about 16 significant decimal digits.

The `double` type gives more precise representations (as well as a larger range)

than the `float` type, and so is more appropriate for scientific calculations. However, **since errors are magnified when calculations are performed, computations with large numbers of floating point calculations mean that unless you are careful, the imprecision inherent in these approximations will lead to large accumulated errors.**

[Footnote: These issues are studied by the field of mathematics known as ***numerical analysis***.]

The default ***floating point literal*** is interpreted as a `double` ; a literal to be treated as a `float` must end with `f` or `F`. (A `double` literal optionally ends with `d` or `D` .)

Floating point numbers can be written using ***decimal notation***, as in the text, or in ***scientific notation*** (e.g., `9.87E-65` or `3.e4`).

The Java ***character*** type is called ***char***. Java characters are represented using an encoding called ***unicode***, which is an extension of the ***ascii*** encoding. Ascii encodes English alphanumeric characters as well as other characters used by American computers using 8 binary digits. Unicode is a 16−bit representation that allows encoding of most of the world's alphabets. ***Character literals*** are enclosed in single quotation marks:

```
'x'
```

Characters that cannot easily be typed can be specified using a ***character escape***: a backslash followed by a special character or number indicating the desired character. For example, the horizontal ***tab character*** can be specified `'\t'`; ***newline*** is `'\n'`; the ***single quote*** character is `'\''`; ***double quote*** is `'\"'`; and ***backslash*** is `'\\'`. Characters can also be specified by using their unicode numeric equivalent prefixed with the `\u` escape.

The true−or−false type is called ***boolean***. There are exactly two `boolean` literals: ***true*** and ***false***.

The names of all Java primitive types are entirely lower case. By convention, we begin each non−primitive type with an upper−case letter.

The double−quoted−sequence−of−characters type is called ***String***. String doesn't actually belong in this list because, unlike the other types listed here, String is not a primitive type. Note that its name begins with an upper case letter. String does have a literal representation, though. (String is the only non−primitive Java type to have a literal representation.) A String literal is enclosed in double quotation marks:

```
"What a String!"
```

It may contain any character permitted in a character literal, including the character escapes described above. For example, the String `"Hello, world!\n"` ends with a newline.

The names of Java primitive types are reserved words in Java. This means that they have special meanings and cannot be used to name other things in Java. (See the sidebar on *Java Naming Syntax and Conventions*.)

3.6.3 Names for Primitive−Type Things: Dial Names

If you want to refer to a Java object, you can do so using an appropriately typed name. You can also refer to a Java primitive−type thing using a name. For example:

```
int myLuckyNumber = 6;
int yourGuess;

yourGuess = myLuckyNumber;
```

The first line creates a new name, *myLuckyNumber*, and binds it to 6. The second line creates a new name, *yourGuess*, but does not explicitly give it any initial value.

[Footnote: In fact, Java provides a default initial value (in this case, 0). However, best programming practice demands that one not rely on such default initializations.]

The final line assigns the value of *myLuckyValue* — 6 — to the name *yourGuess*.

So far, names with primitive type look a lot lot object names. But it turns out that there are some important, if subtle, differences. Java names with primitive types aren't exactly labels, as object names are. You see, there may be an unpredictably large number of Buttons or KlingonStarships, and so a label is the best way to keep track of any particular KlingonStarship that comes along. But Java primitives are different.

For example, there are only two `boolean` values possible in Java: `true` and `false`. If I have two identical−looking KlingonStarships, they are still different ships. Blowing up one doesn't blow up the other. But `booleans` are different. There is really only one `boolean` `true`. Of course, there's not much that you can do with `true` — you can't *change* `true`, the way you can repaint a KlingonStarship — so it's hard to see that there is only one `true`. But there is.

This means that we can use a very different mechanism for a name that has type `boolean`. We can, for example, use a switch as the name. Let's say we have a

`boolean` name, *isSunny*. This name is just a switch. It is always set in one of the two positions: *on/true*, or *off/false*. So if the switch corresponding to *isSunny* is *on*, we'll know that *isSunny* is `true`. If the *isSunny* switch is *off*, we'll know that *isSunny* is `false`. We can tell the value corresponding to the `boolean` name just by inspecting the switch.

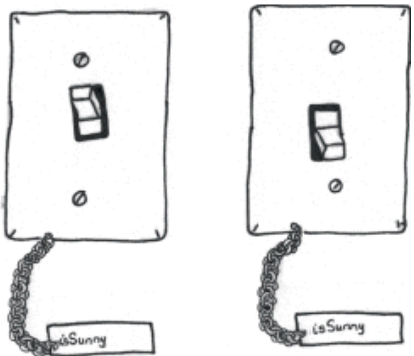


Figure 3.7. A `boolean` name (here, *isSunny*) is just a *switch*.
On the left, the switch is on (true); on the right, the same switch is off (false).

But if we use a switch to indicate true−or−false, how do we do assignment? When we use label names — for objects — we just stick a second label on the same object. When we use ***switches as names***, though, creating a new name means creating a new switch. So what do we do when, for example, we have

```
boolean amHappy = isSunny;
```

(which means that I am happy if it's sunny, and not happy if it's not)? Simple enough: we set the new switch — *amHappy* — to the same position that *isSunny* is in. (Note: We do this once, at the time of the assignment. After that, the two switches are completely separate. More on this later in this section.)

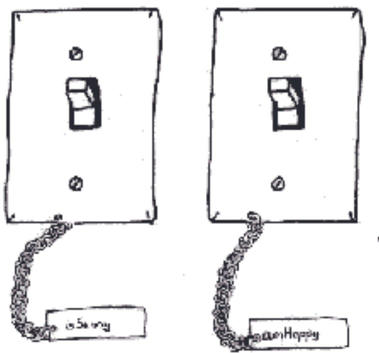


Figure 3.8. Copying values from one switch to another.

It turns out that this analogy works for all of the Java primitive types. For example, there are only a fixed, finite number of `ints` possible in Java. (See the sidebar on *Java*

Primitive Types.) Although it might be confusing to imagine a switch with 4,294,967,296 positions, you can imagine that an `int` name is just a very large ***dial*** with those same 4 billion settings. By reading the setting of the dial, you can tell what value an `int` name corresponds to.

As remarkable as it may seem, each of the Java primitive types is represented in this way. The dial here is metaphorical, but the actual representation is very much as described. A Java primitive is stored in such a way that every name with a primitive type indicates its value just as the metaphoric dial does. A name corresponding to a ***byte*** is simply a dial with 256 positions. A ***long*** name has 2^{64} positions. And even ***float*** and ***double*** names have only finite numbers of positions; this is why ***floating point numbers*** don't *really* represent ***real numbers*** and in fact aren't all that good for extreme precision calculations.

[Footnote: Even a double precision floating point number can represent only 2^{64} values between -1.8×10^{308} and 1.8×10^{308}, so many values just can't be captured accurately. Note also that the actual precision of a `double` varies over this range.]

The `char` type ranges over many more values than just *a* through *Z*, but there are still only 2^{16} possible characters in Java, so a `char` name is like a dial with that many positions.

For example,

```
int i;
```

associate *i* with a dial that's just the right size for a 32−bit integer.

How is this different from a label? There are at least three big differences.

1. **Labels can be `null`; dials cannot.**

   A label can be `null`, meaning that it is not “stuck” to any object. A dial cannot be `null`; by the very nature of a dial, its hand is on *some* value.

2. **When a type X label is declared, no object of type X is created; when a type Y dial is declared, the dial acquires a type Y default value.**

   When a label of a certain object−type is declared, a label name appropriate for that object−type is created (and storage allocated for it), but no object of that object−type is created. When a dial of a certain primitive−type is declared, a dial name appropriate for that primitive−type is created and also, by the very nature of a dial, its hand is set to *some* value of that primitive−type.

3. **Labels can “share” a value; dials cannot.**

Two labels can be stuck on to the same object; then, changing the object by using the first label to reference it changes the (same) object referenced by the second label. While two dials can have the same value (both are set to the same position on the dial), the dials are independent — changing the setting on one dial never changes the setting on another dial.

In particular, assigning one label to another means that both are stuck on the same object (or both are null), so that they "share" a value. When one dial is assigned to another, the setting on the latter dial is copied onto the former. That's it; after that, the assignment is complete and the two dials go their separate ways. There is no further relationship between the values on the dials.

The last point is worth a closer look. Consider the following two very similar−looking sets of statements (one on the left, the other on the right):

```
int i;                    |         Cat marigold;
i = 3;                    |         marigold = new Cat();
                          |
int j = i;                |         Cat phoebe = marigold;
                          |
i = 4;                    |         marigold.haveKittens();
```

In both cases, the first statement declares a name and the second statement assigns the name a value. The name *i* on the left is a dial−name and is set to 3; the name *marigold* on the right is a label−name and is set to a new Cat.

In both cases, the third statement declares another name and assigns the first name to the second. However, assignment has a different meaning in these two cases. On the left, the dial−name *j* is set to the same setting (3) as the dial−name *i*. On the right, the label−name *phoebe* is stuck onto the same object that the label−name *marigold* is stuck on.

In both cases, the fourth statement changes something by using the first name (*i* on the left and *marigold* on the right). However, the effects are quite different. On the left, the value of *i* changes to 4, of course, but the value of *j* remains 3. If we asked whether *j* is 4, the answer would be "No." On the right, the object referenced by *marigold* has kittens, so that (same) object referenced by *phoebe* has kittens. If we asked whether *phoebe* has had kittens, the answer would by "Yes!"

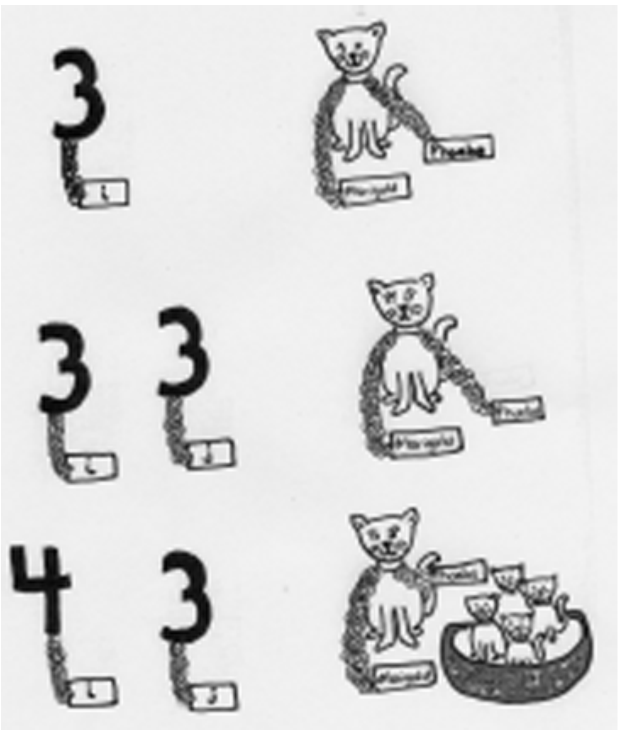


Figure 3.9. Dial names versus label names: the effect of assignment.
The left side of the figure shows the **dial** names *i* and *j*.
The right side of the figure shows the **label** names *marigold* and *phoebe*.
The middle row shows the different effects of assignment in the two cases.

To summarize:

- When you are using names that are declared to be of object−types, think of those names as labels that can be stuck on objects of the declared type.

- When you are using names that are declared to be of primitive−types, think of those names as dials that always are set to some physical point.

- This implies three key distinctions:

  - ♦ **Labels can be `null`; dials cannot.**

  - ♦ **Declaring a label does NOT create an object of the declared type.**

  - ♦ **Labels can “share” a value; dials cannot.**

- It is easy to recognize primitive−types: they are all lower−case letters, while (by convention) we always use an upper−case letter as the first letter of any object−type.

The dial and label metaphors that we have used here, while helpful, are not standard terminology. Instead of *labels* you will often see the phrase ***reference−type names***. Instead of *dials* you will often see the phrase ***value−type names***.

3.7 A Tale of Things and Names

To conclude this chapter, let's walk through an example of how things are named and how names refer to things. This example involves a story that took place some years ago at a fancy party, the kind of party where one leaves one's hat at the door and whose attendees might include a few famous names...

This story concerns one such party and three such personages: Charlie Chaplin, King George VI of England, and Eleanor Roosevelt.

First among our characters to arrive was Charlie Chaplin. He was wearing his usual bowler hat.

```
Hat charlieChaplinHat;
```

The code above just tells us that *charlieChaplinHat* is a label suitable for naming a hat, but we will also assume that the *charlieChaplinHat* label is, as usual, stuck on the bowler:

Figure 3.10. The binding of *charlieChaplinHat* to Charlie Chaplin's bowler.

When Chaplin arrived in the lobby, he saw the hat check. He took off his hat and handed it to the hat check. Fortunately the hat check had a *checkHat* method, requiring a Hat:

```
hatCheck.checkHat(charlieChaplinHat);
```

Of course, now Charlie Chaplin wasn't wearing a hat:

```
charlieChaplinHat = null;
```

So unburdened, he walked in to the party.

Next to arrive was the King of England. He was wearing his crown.

```
Hat kingGeorgeHat;
```

Again, we assume that — prior to the execution of the rest of this code — *kingGeorgeHat* is labeling the crown.

When King George arrived at the hat check, he, too, removed his hat and gave it to the hat check.

```
hatCheck.checkHat(kingGeorgeHat);
kingGeorgeHat = null;
```

[Footnote: **Question:** What would have happened if the King had executed these lines in the opposite order:

```
kingGeorgeHat = null;
hatCheck.checkHat(kingGeorgeHat);
```

]

Then he, too, went in to the party. Both men had a lovely evening at the party. Mr. Chaplin left first. Reentering the lobby, he approached the hat check and observed that it also provided a *returnHat* method. So he executed:

```
charlieChaplinHat = hatCheck.returnHat();
```

Much to his surprise, the hat handed to him by the hat check was not his simple black bowler. Instead, it was the magnificent crown of the king of England. Chaplin was still staggering around the lobby under the weight of his crown when King George emerged from the party and approached the hat check.

```
kingGeorgeHat = hatCheck.returnHat();
```

I'm sure it won't surprise you to hear that the King now found himself in possession of the simple black bowler. Being a man of simple good taste and few pretensions, George was half tempted to leave with the hat, but it occurred to him that there might be some other gentleman who would then regret the loss of such a sturdy topper. Further, he knew well from experience that the crown of England can be a heavy burden and he hated the thought that someone else might have to suffer under it. Turning, he saw Charlie Chaplin staggering under just that burden.

“Sir,” King George observed, “It seems our headgear has been exchanged by the hat check. Perhaps we should remedy this situation.”

And so the two men proceeded to try just that. But there was a problem. Between them,

the men had only two hat labels, and each was occupied. For example they considered executing

```
kingGeorgeHat = charlieChaplinHat;
```

but this would have resulted in both men holding the crown, and Mr. Chaplin's trustee bowler lost in limbo with no way to retrieve it (since it would have been labelless). The two men were puzzling over this dilemma when who should emerge from the party but Eleanor Roosevelt. (She'd snuck in while the reader wasn't looking.) This savvy diplomat immediately took measure of the situation.

“I see that you two gentlemen have run into a bit of difficulty. Perhaps I can be of some assistance. You see, I am not wearing a hat, and so my head can be a temporary resting place while the two of you work your exchange.

“First we will need a label for my hat:

```
Hat eleanorRooseveltHat;
```

“You see, of course, that there's no Hat there: *eleanorRooseveltHat* is `null`. It's simply a label that *could* be stuck on a Hat; it's the potential Hat−holder representing my head.”

[Footnote: Actually, what Mrs. Roosevelt really said was, “As you can see, **`eleanorRooseveltHat == null`** is now true, but of course you won't be introduced to **`==`**, the identity operator, until Chapter 5.”]

Next, the erstwhile Mrs. Roosevelt offered to take the crown of England from Mr. Chaplin, to which he readily agreed:

```
eleanorRooseveltHat = charlieChaplinHat;
```

Now, the tall thin diplomat and the short comic actor found themselves jointly holding England's crown. At this, Mrs. Roosevelt suggested that Mr. Chaplin release the crown by seizing instead the simple black bowler to which he was accustomed (and of which the king of England was growing overly fond).

```
charlieChaplinHat = kingGeorgeHat;
```

Mrs. Roosevelt now had sole possession of the crown; Mr. Chaplin and King George both held the bowler. At this, King George released the bowler to take possession of his crown:

```
kingGeorgeHat = eleanorRooseveltHat;
```

Finally, Mrs. Roosevelt released her hold on the crown, freeing her to return to important business:

```
eleanorRooseveltHat = null;
```

(Of course, this step was not strictly necessary as each gentlemen now held the proper hat. Mrs. Roosevelt simply wanted to leave open her options for wearing other hats later.)

And the three figures left the party satisfied that all was well.

Chapter Summary

- *Literals* are things you can type directly to Java.
- Java has eight primitive types:
  - `char` is the type for single keystrokes (letters, numbers, etc.)
  - `int` is the standard type for integers. Other integer types include `byte`, `short`, and `long`.
  - `double` is the standard type for *floating point numbers*, which are approximations to real numbers. The `float` type is another floating−point type.
  - `boolean` is a type with only two values, `true` and `false`.
  - The limited range of all the numeric types means that calculations using such numbers can *overflow* and give wrong results. The limited precision of the floating−point types means that repeated calculations involved them can lead to large accumulated errors. The programmer must be aware of such differences between computer arithmetic and real arithmetic.
- All other Java types are object types.
  - All eight of the primitive types begin with a lower−case letter.
  - By convention, we begin each object type with an upper−case letter.
- String is the type for arbitrary text (sequences of characters). String is not a primitive type, but Java does have String literals.
- Names can be used as placeholders for values. Every name is born (declared) with a particular type, and can label only things having that type.
- Primitive types have dial names. A dial name always has an associated value. Two dials cannot share a single value; each has its own copy.
- Object types have label names. Two label names can label the same object. A label that is not currently stuck on anything is associated with the non−value *null*.

Exercises

1. Assume that the following declarations apply:

```
int i;
char c;
boolean b;
```

For each item below, give the type of the item.

a. `42`
b. `−7.343`
c. `i`
d. `'c'`
e. `“An expression in double-quotes”`
f. `b`
g. `false`
h. `“false”`
i. `c`
j. `'b'`
k. `“b”`

2. For each of the following definitions, fill in a type that would make the assignment legal.

[Footnote: There are several answers to some of these, but in each case only one “most obvious” type. It is this “most obvious” type that we are after.]

```
__________ a = 3;

__________ b = true;

__________ c = 3.5;

__________ d = “true”;

__________ e = “6”;

__________ f = null;

__________ g = 0;

__________ h = '3';

__________ i = '\n';

__________ j = “\n”;
```

3. This problem checks your understanding of *assignment*.

a. Assume that the following statements are executed, in order.

```
int a = 5;
int b = 7;
int c = 3;
int d = 0;

a = b;
c = d;
a = d;
```

What is the value of **a**? of **b**? of **c**? of **d**?

b. Assume that the following statements are executed, in order.

```
int a = 5;
int b = 7;
int c = 3;
int d = 0;

a = b;
b = c;
```

What is the value of **a**? of **b**? of **c**?

c. Assume that the following statements are executed, in order.

```
char a = 'a';
char b = 'b';
char c = 'c';
char d = 'd';

a = b;
c = a;
a = d;
```

What is the value of **a**? of **b**? of **c**? of **d**?

d. Assume that *myObject* is a name bound to an object (i.e., *myObject* is not *null*). After the following statements are executed in order,

```
Object a = myObject;
Object b = null;
Object c = a;

a = b;
```

Is the value of **a** *null* or non−*null*? What about **b**? What about **c**? What about **myObject**?

e. Assume again that *myObject* is a name bound to an object (i.e., *myObject* is not *null*). After the following statements are executed in order,

```
Object d = myObject;

d = null;
```

Is the value of **d** *null* or non−*null*? What about **myObject**?

f. Assume one more time that *myObject* is a name bound to an object (i.e., *myObject* is not *null*). After the following statements are executed in order,

```
Object e = myObject;

myObject = null;
```

Now is the value of **e** *null* or non−*null*? What about **myObject**?

4. Which of the following could legitimately be used as a name in Java? (Note that *none* of them would be wise choices for names, except possibly in a Star Wars game, as none of them is likely to convey meaningful information to readers of a program.)

```
3PO                        R2D2
c3po                       luke
jabba_the_hut              PrincessLeia
Han Solo                   obi-wan
foo                        int
Double                     character
string                     goto
elseif                     fi
```

5. Assume that the following declarations have been made:

```
int i = 3;
int j;
char c = '?';
char d = '\n';
boolean b;
String s = "A literal";
String s2;
Object o;
```

Complete the following table:

Name	dial or label?	Value (or `null`)?
i		
j		
c		
d		
b		
s		
s2		
o		

6. Assume that there is an already−defined object type called *Date* and that *today* is an already−defined Date name with a value representing today's date. Suppose that you wanted to declare a new name, *yesterday*, and give it the value currently referred to by *today*. This would be useful, for example, if it were nearly midnight and we might soon want to update the value referred to by *today*.

Explain why the following attempt will *not* successfully solve this problem.

```
Date yesterday;

yesterday = today;
```

[Footnote: At this point of this book, you should understand why the above will *not* work, but we have not yet discussed what *would* work. We'll see the basic ideas for a successful solution in Chapter 7, *Building New Things: Classes and Objects*. Also relevant is the discussion about *clone* and Cloneable in Chapter 10, *Inheritance*.]

7. Continuing the previous problem, now assume that *today* is an already−defined `int` (not Date) name with a value representing today's date, where 1 represents January 1, and 32 represents February 1, and 33 represents February 2, and so on, with 365 representing December 31. (Assume that this is not a leap year.) Again suppose that you wanted to declare a new name, *yesterday*, and give it the value

currently referred to by *today*. Now, the problem is solvable with the tools from this chapter.

Give the solution (that is, declare *yesterday* appropriately and give it the value referred to by *today*). Then explain how this problem is different from the previous problem.

8. Recall again the tale of two hats. Later, after the events of that story were long past, someone suggested that the problem here was that the hat check hadn't issued claim checks, which might have changed the circumstances. These claim checks would have been `int`s, not Hats.

   "If the names had been dials rather than labels, would Mrs. Roosevelt's assistance still have been needed?"

   As it happens, there *were* claim checks involved, but Charlie Chaplin and King George had clumsily dropped them, and they were unable to determine which claim check was whose. That is why they'd each used the *hatCheck.returnHat* method, rather than a method that required a claim check as input.

   Suppose that the situation were as follows:

   After the mixup, Mr. Chaplin found himself holding claim check 2:

   ```
   int charlieChaplinCheck = 2;
   ```

   while King George was in possession of claim check 1:

   ```
   int kingGeorgeCheck = 1;
   ```

   However, the crown can be retrieved only with claim check 2, the bowler only with claim check 1. So the two gentlemen are now faced with a swap of integers rather than hats. As the story originally unfolded, they gave up and later were forced to proceed with the hat swap. Imagine, instead, that Mrs. Roosevelt had walked up at that moment — when the claim checks needed rearranging.

   Write code to resolve the situation *without* using the literals `1` and `2` further. You may, of course, use *charlieChaplinCheck* and *kingGeorgeCheck* in your code, as well as Mrs. Roosevelt.

Chapter 4

Specifying Behavior: Interfaces

Chapter Overview

- How do programs (and people) know what to expect?
- How do I describe a part or property of an entity to other community members?

This chapter introduces the idea of ***interfaces*** as partial program specifications. An interface lets community members know what they can expect of one another and what they can call on each other to do; in other words, interfaces specify "how they interact." In this way, an interface describes a contract between the provider of some behavior and its user. For example, the post office promises to deliver your letter to its intended recipient if you give it to them in the appropriate form. This promise (together with its requirements for a properly addressed and stamped envelope, etc.) constitutes a part of the post office's interface.

In this chapter, you will learn how to read and write Java interfaces. These allow you to use code designed by others — in the same way that you can drop off an appropriately addressed letter at the post office — and to tell others how to use the services that you provide. You will also learn about things that an interface *doesn't* tell you. For example, when you drop a letter off at the post office, you don't necessarily know whether it's going by truck or by train to its destination. You may not know when it is going to arrive. This chapter concludes with a discussion of what isn't specified by an interface and how good documentation can make some of these other assumptions explicit.

This chapter is supplemented by a reference chart on the syntax and semantics of Java interfaces.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Objectives of this Chapter

1. To learn how to recognize and read Java *method signatures*.

2. To understand how an *interface* specifies a contract between two entities while separating the user from the implementation.

4.1 Interfaces are Contracts

Programs are communities of interacting entities. How does one entity know what kinds of services another entity provides? How do programmers know what kinds of behavior they can expect from objects and entities that they haven't built? A key to understanding these questions is the notion of *interface*.

An ***interface*** is a contract that one object or entity makes with another. Interfaces represent agreements between the ***implementor*** (or builder) of an object and its ***users***. In many ways, these are like legal contracts: they specify some required behavior, but not necessarily how that behavior will be carried out. They also leave open what other things the parties to the contract may be doing.

An excellent example of a standardized interface is an electrical outlet. In the United States, there is a particular standard for the shape, size, and electrical properties of wall outlets. This means that you can take almost any US appliance and plug it in to almost any US wall outlet and rest assured that your appliance will run. The power company doesn't need to know what you're plugging in — there are no special toaster outlets, distinct from food processor outlets, for example — and you don't need to know whether the power company produced this electricity through a hydroelectric plant or a wind farm. The outlet provides a standard interface, with a particular contract, and as long as you live within the parameters of that contract, the two sides of the interface can remain relatively independent.

Of course, there are places where this contract breaks down. US appliances don't generally work in European outlets, for example. There are several standard electrical outlet interfaces throughout the world. It isn't clear that one of them is particularly better than another, but it is unquestionably true that you can't use one side of the US outlet interface (e.g., a US appliance) with the other side of the European interface (a 220V outlet). If you want to mix and match disparate interfaces, you will need a special adapter component. The same is true for software.

There are also, even in the US, certain appliances that can't use standard wall outlets. For

example, an electric oven draws too much current, and so needs a special kind of wall outlet. The physical connector — the plug — is different on this appliance, to indicate that it fits a different interface. You can't plug an electric oven in to a standard US wall outlet. This is because its needs don't meet the (sometimes implicit) constraints of standard (15 or 25 amp) US circuits. Sometimes this happens in software, too — you need a different interface because the standard one doesn't provide precisely the functionality that you need.

4.1.1 Generalized Interfaces and Java Interfaces

The dictionary defines ***interface*** as "the common region of contact between two independent systems." In Computer Science, we use interface to mean the boundary between two (or more) things. In general, when you are constructing a community of interacting entities, interface refers to the "face" that one of these entities shows another: what services it provides, what information it expects. One entity may, of course, have many interfaces, showing different "faces" to different community members.

Interface is a piece of the answer to the question of how things interact.

User interface refers to the part of a computer program that the person using the computer actually interacts with. For example, a ***graphical user interface (GUI)*** is one that uses a certain interaction style, e.g., typically contains buttons and menus and windows and icons. (Before GUIs, computer interfaces typically used text, one line at a time, the way that some chat programs work now.) A good user interface takes into account the properties of the program as well as those of human users. Not surprisingly, humans and computers have different skill sets. Like user interfaces, every interface should be designed bearing in mind the needs of the entities on both sides. We will learn more about graphical user interfaces in particular in Parts 3 and 4 of this book.

This Computer Science use of the word *interface* is one sense in which we will use the term in this book. In Java, there is a second, related but much more limited use of the word *interface*. A ***Java interface*** refers to a particular formal specification of objects' behavior. The keyword `interface` is used to specify the formal declaration of a particular kind of contract guaranteeing this behavior. (For example, there might be an interface defining clock−like behavior.) The Java language defines the rules for setting out that contract, including what can and can't be specified by it. A particular Java interface is a particular promise.

In this book, when we use the term *Java interface* or the code keyword `interface`, we are referring to this formal declaration. When we use the term *generalized interface*, we are referring to the more general Computer Science notion of interfaces. A Java interface is one way to (partially) specify a generalized interface. There may be things that are part of the general promise — such as how long a particular request might take to answer — that can't be specified in a Java interface.

This chapter deals specifically with Java interfaces. The ideas of generalized interfaces permeate all parts of this book; the generalized notion of an interface is central to interactive program design. We will explicitly revisit this issue — generalized interface design — in the chapters on Protocols and Communication in Part 5 of the book.

4.1.2 A Java Interface Example

Consider, for example, a ***counter*** such as appears on the bottom of many web pages, recording the number of visitors. Most such ***Counting*** objects have a very simple interface. If you have a Counting object, you expect to be able to ***increment*** it — add one to the number that the Counting object keeps track of — or to be able to ***read*** — or get — its current value. This is true pretty much no matter how the Counting object actually works or what other behavior it might provide. In fact, by this description, a stopwatch might be a special kind of Counting object that automatically increments itself. So we might say that *increment* and *getValue* form a useful interface contract specifying what a (minimal sort of a) Counting object might be. In Java, we write this as:

```
interface Counting {       // gives the name of the interface
    void increment();      // describes the increment contract
    int getValue();        // describes the getValue contract
}
```

We will see below how to read this interface declaration.

Once you and I agree on an interface for a Counting object, I can build one and you can use it without your needing to know all of the details of how I built it. You can rely on the fact that you will be able to ask my Counting object for its current value using *getValue*. Your code, which uses my Counting object, doesn't need to know whether *increment* adds one point (for a soccer goal) or six (for a touchdown in American football). It doesn't need to know whether I represent the current value internally in decimal or binary or number of touchdowns, field goals, etc.

Your code should work even if I exchange my original Counting object for one that can be reset before each game or each time I rewrite my web page, since your code depends only on being able to increment and read the value of my Counting object. In turn, I can go off and build a Counting object using whichever internal representations I wish to provide, so long as I meet the contract's commitments (*increment* and *getValue*).

Of course, you may want to know more about my Counting object than what the *increment/getValue* parts of the Counting interface tells you. Some of this information may be contained in the documentation for Counting. (This Counting object's value will always be non−negative.) Other information may be contained in the documentation for my particular implementation. (My ***BasicCounter*** implementation of my Counting object is guaranteed to increase; its value cannot decrease.) If you want to know whether my

clock provides additional services, though, you may need to use an interface that specifies this additional behavior (e.g., a Resettable interface). We will discuss the kinds of information conveyed by an interface, and that which should be included in interface documentation, later in this chapter.

4.2 Method Signatures

In the StringTransformer interlude and briefly in the discussion of objects, we have seen ***methods***, behavior that objects provide. These methods are essentially rules for how to accomplish particular behaviors. In an interface, we focus on the specifications for these rules and not on the instructions for how to achieve them. That is, **an *interface* is a collection of *rule specifications*. Any object that implements that interface must satisfy those specifications, though there are virtually no limits on how it might do that.**

The formal name for a rule specifications is a ***method signature***. For example, the Counting interface specifies two rules — *increment* and *getValue* — that every Counting object must provide. The body of the interface declaration is these two method signatures, or rule specifications. A method signature describes what things that rule expects (or needs to know about) and what the rule will return. It also needs a name, so that you can refer to and invoke the rule (of course). In Chapter 11, *Exceptions*, we will see that there is one other kind of thing that can be a part of a rule specification.

Unlike the method itself, a method signature does *not* need a body. The ***body*** is the part of the method (or rule) that contains the instructions specifying how to do the behavior, and that is not a part of the interface/promise. The rule specification is only that part of the promise that users of the object need to know: what request to make, what things to give the rule, and what to expect back. The rule body — how to do the rule — is needed only by the rule implementor, not by the rule user.

In the particular case of the Counting interface, there are two rules that every Counting object must implement: *increment* and *getValue*. So the Counting Java interface would need to specify these two method signatures.

Each method signature has three parts: *name*, *parameter specification*, and *return type*. The next three subsections describe, for each of these three parts, both the obligations of the designer of the interface and the ways in which the interface is used by another entity.

[Footnote: There is actually one other part of some method signatures, the ***throws*** clause. Every method signature must have a name, parameter list, and return type, but some methods do not have a `throws` clause. The `throws` clause will be introduced in Chapter 11, *Exceptions*. In addition, certain modifiers — such as `abstract`, explained below — may be included in a method signature.]

4.2.1 Name

When you are *designing* an interface, a rule can have any name that you want to give it. It is a good idea to give it a name that will help you (and the users of your code) remember what it does. Remember the *syntax of Java names* — alphanumeric and a few symbolic characters — from Chapter 3, *Things, Types and Names*, and that rule/method names should start with a lower−case letter.

When you are *using* an interface, the name of the rule is whatever name the interface says it is. Hopefully, the name was chosen well so that it is easy to remember and to figure out what that rule does.

4.2.2 Parameters and Parameter Types

These are the things that your rule needs in order to work. (For example, the StringTransformer's *transform* rule needs to know what String to transform.) A ***parameter*** is a temporary name associated with a value supplied when the method is ***called*** , i.e., when the rule that it represents is ***invoked***. During the execution of the rule, the parameter name can be used to refers to the supplied value.

When you are designing an interface, you will need to specify a type and a name for each parameter. (The *Type−of−thing Name−of−thing* rule from Chapter 3, *Things, Types, and Names*, strikes again.) The type can be any legal Java type (including both primitive and object types); the name can be any Java−legal name that you choose to give the parameter. It is advisable that you give your parameters names that make it easy for the users and implementors of your rule to figure out what role the particular parameter plays in the rule. Our convention is to use names that begin with a lower−case letter for parameters.

The list of parameters is separated by commas:

```
Type-of-thing  Name-of-thing,  Type-of-thing  Name-of-thing, ...
```

and so on until the last *Type−of−thing Name−of−thing*, which doesn't have a comma after it. The whole list is enclosed in parentheses. You can list your parameters in any order. Of course, some orders will naturally make more sense than others, and although the choice is arbitrary, once chosen the order is fixed. This means that users and implementors of the method will need to follow the order declared in the interface.

The *getValue* and *increment* rules of Counting don't have any parameters, i.e., they don't need any information to begin operation. Their parameter lists are empty: **`()`**. StringTransformer's *transform* rule needs one parameter, a String. We can call that String anything we want to. For example, *transform*'s parameter list might be:

```
(String whatToTransform)
```

A more complex ***AlarmedCounting*** interface might be mostly like our Counting interface but in addition have a *setAlarm* method that takes two parameters, one an `int` indicating the value at which the alarm should go off and the other a String that should be printed out when the alarm is supposed to be sounded:

```
setAlarm(int whatValue, String alarmMessage)
```

When you are using a method, you need to pass the method a set of ***arguments*** that match the parameter list. That is, between the parentheses after the name of the method you're invoking, you need to have an expression whose type matches the type of the first parameter, followed by a comma, followed by an expression whose type matches the type of the second parameter, and so on, until you run out of parameters, e.g.:

```
increment()
transform("a string to transform")
setAlarm(1000, "capacity exceeded")
```

4.2.3 Return Type

The rule also needs to specify what its users can expect to get back. In many cases, the rule returns a value. The ***return type*** is then the type of the value returned. In some cases, the rule does not return a value. (The *increment* method is an example of such a rule: it changes the value stored inside the Counting object, but doesn't give anything back to the entity that invoked it.) The return type of such a rule is a special Java keyword: `void`. The only purpose for `void` is as the return type of rules that don't return a value. The Counting interface's *increment* method presumably doesn't return anything, so its return type would be `void`. The return type of the *getValue* method is presumably `int`.

When you use a method, you may or may not want to do something with the value returned. The return type of the method signature tells you what type of thing you can expect to get back, e.g., so that you can declare an appropriate name to store the result:

```
int counterValue = myCounting.getValue();
```

where *myCounting* is something that implements the Counting interface, i.e., satisfies the Counting contract (and therefore has an `int`-returning *getValue* method). After this statement, *counterValue* is a name that refers to whatever `int` *myCounting*'s *getValue* method returned.

4.2.4 Putting It All Together: Abstract Method Declaration Syntax

Now you know about all of the components of a method signature. All you need to know is how to put them together. The *Type-of-thing Name-of-thing* rule from Chapter 3, *Things, Types and Names*, comes into play here as well. The ***type of a method*** is its return type, so a ***method specification*** is:

```
returnType ruleName(paramType1 paramName1, ... paramTypeN paramNameN);
```

For example,

```
int getValue();
```

or

```
void increment();
```

Note that these declarations end with a semi−colon (`;`). This means that the method signature is being used here as a specification — a contract. It doesn't say anything about how the method — say *increment* — ought to work. That is, it doesn't even have a space for the rule body, just the rule specification.

This form — method signature followed by a semi−colon — is called an ***abstract method***. There is even a Java keyword — `abstract` — to describe such methods. It is OK, if sometimes redundant, to say

```
abstract void increment();
```

instead of the form given above. This is different from the use of a method signature together with its body to define behavior (i.e., in a class declaration). We will see how to use method signatures in the declaration of classes in Chapter 7, *Building New Things: Classes and Objects*.

Since interfaces always specify only method signatures, interface method declarations are always `abstract`. If you don't say so explicitly, Java will still act like the word `abstract` is there. However, if your method definition does not end with a semi−colon, your Java interface will not compile.

4.2.5 What a Signature Doesn't Say

The properties of a method that are documented by its signature are its name, its parameters, and its return type.

[Footnote: In addition, method signatures may include visibility and other modifiers and any exceptions that the method may throw.]

That leaves a whole lot open.

For example, for each parameter:

- What is that parameter intended to represent?
- What relationships, if any, are expected to exist among the parameters?

- Are there any restrictions on the legal values for a particular parameter?
- Will the object represented by a particular parameter be modified during the execution of the method?

For the return type:

- What is the relationship of the returned object to the parameters (or to anything else)?
- What may you do with the object returned? What may you not do?

Other questions not included in the method signature:

- What preconditions must be satisfied before you invoke this method?
- What expectations should you have after the method returns?
- How long can the method be expected to take?
- What other timing properties might be important?
- What else can or cannot happen while this method is executing?

Not all of these questions are relevant to every method. For example, the precise amount of time taken by the Counting object's *getValue* method is probably not important; it is important that it return reasonably quickly, so that the value returned will reflect the state at the time that the request was made. However, it is important to recognize that these and other questions are not answered by your method signatures alone, so you must be careful to ***document*** your assumptions using Java ***comments***.

Style Sidebar

Method Documentation

Documentation for a method should always include the following items:

- Why would you want to use this method? What does it do? When is it appropriate (or not appropriate) to use this method? Are there other methods that should be used instead (or in addition)? Are there any other “hidden assumptions” made by this method?
- What does each parameter represent? Is it information supplied by the caller to the method? Is it modified during the execution of the method? What additional assumptions does the method make about these parameters?
- What does the return value of the method represent? How is it related to the method's arguments or other Things in the environment? What additional assumptions may be made about this return value?
- What else might be affected by the execution of this method? Is something printed out? Is another (non−parameter) value modified when it is run? These non−parameter non−return effects are called ***side effects***.

In addition, if there are other assumptions made by the method — such as how long it can take to run or what else can (or cannot) happen at the same time — these should be included in the method's documentation.

Java provides additional support for some of these items in its ***Javadoc*** utilities. See the appendix on *Javadoc* for details.

4.3 Interface Declaration

Now that we know all about Java method signatures, it is very easy to declare a Java interface. A ***Java interface*** is simply a collection of method signatures.

4.3.1 Syntax

A Java interface is typically declared in its very own file. The file and the interfaces generally have the same name, except that the file name ends with `.java`. (For example, the Counting interface would be declared in a file called *Counting.java*.)

Like most other declarations, an interface follows the *Type−of−thing Name−of−thing*

rule. The *type−of−thing* is, in this case, `interface`. The name is whatever name you're giving the interface, if you're declaring it:

```
interface Counting
```

Now comes an ***interface body***: an open−brace followed by a set of method signatures followed by a close−brace. Note that it doesn't matter in which order the two methods are declared; the two possible orders are equivalent. The whole thing (including the **`interface Counting`** part) looks like this:

```
interface Counting {
    abstract void increment();
    abstract int getValue();
}
```

That's all there is to it.

Question: In this definition of Counting, the word `abstract` appears twice. In the previous definition, above, it doesn't appear at all. Explain.

In fact, that was so easy, let's try another interface. This one is ***Resettable***, and it is a very simple interface. (Good interfaces often are.) Resettable has a single method:

```
interface Resettable {
    void reset();
}
```

This interface is fine, but it could do with a little bit of documentation. After all, there are many things that an interface *doesn't* specify. **Question:** Can you identify some things that should be included in Resettable's documentation?

For the precise specification of what may be included in an interface definition, in what order, and under what circumstances, see the *Java Chart on Interfaces*.

4.3.2 Method Footprints and Unique Names

It might seem that each method in an interface would have a unique name. However, it turns out that this isn't the case — at least, not exactly. Instead of a unique name, each method in an interface (or class) definition must have a unique ***footprint***. The method's footprint consists of its name *plus* its ordered list of parameter types. Only the ordered list of parameter types counts; the return type of the method, and the names given to the parameters, are not relevant to its footprint.

For example, a *reset* rule with no parameters (an empty parameter list, `()`) has a different footprint from a

```
reset(int newValue)
```

rule (with the parameter list `(int)`), and both are different from

```
reset(String resetMessage)
```

(parameter list (String)). Only the parameter type matters, though, not the parameter names:

```
reset(String resetMessage)
```

is the same as

```
reset(String whatToSay)
```

As long as two methods have different footprints, they can share the same name. This is very common and even has its own name: ***overloading***. Overloading allows an object to have two (or more) similar methods that do slightly different things. For example, there are two very similar mathematical rounding methods. One has the signature:

```
int round(float f);
```

while the other has the signature:

```
long round(double d);
```

The Math object has both of these methods, and if you pass *Math.round* a `float`, you get back an `int`, while if you pass it a `double`, you get back a `long`. This is very convenient — in both cases, a floating point number is converted to an integer, but in either case the more appropriate size is used.

An alternate kind of overloading might happen if our hypothetical AlarmedCounting interface had, in addition to its

```
void setAlarm(int whatValue, String alarmMessage)
```

method, a second method that just allowed you to specify the alarm message, without changing the value for which it was set:

```
void setAlarm(String alarmMessage)
```

If you called

```
yourAlarm.setAlarm(1000, "Capacity reached")
```

you'd set the alarm message to trigger at 1000, printing the message “Capacity reached,” while

```
yourAlarm.setAlarm("Oops, all full")
```

might then be used to change the warning to be issued when the AlarmedCounting reaches capacity.

Overloading method names is the choice of the interface *builder*. The interface *user* simply makes use of the interface as it is given.

4.3.3 Interfaces are Types: Behavior Promises

Now that we have these interfaces, what good do they do? Interfaces are *kinds of Things*: they are Java types.

In Java, **every interface name is automatically a type name.** That is, when you are declaring a (label) name, you can declare it suitable for labeling things that implement a specific interface. In Chapter 7, *Building New Things: Classes and Objects*, we will see how to declare Java classes and how to indicate what interface(s) the class implements.

So, for example, the declared type of *myCounting*, above, was Counting:

```
Counting myCounting;
```

In this example, *myCounting* is declared to be of type Counting, i.e., something that satisfies the Counting contract (interface) that we declared in the preceding sections. For example, we might have an interface called Game that includes a *getScoreCounter* method that returns a Counting:

```
interface Game {
    Counting getScoreCounter();
    // maybe some other method signatures....
}
```

If *theWorldCupFinal* is a Game, then we might say

```
Counting myCounting = theWorldCupFinal.getScoreCounter();
```

In this case, we don't know anything more about the type of *myCounting*; we just know that it is a ClassCounting. Often, as users of other people's code, interfaces are the only types we need to know about.

4.3.4 Interfaces are Not Implementations

We have seen that an interface can be used as the type of an object. You can use names associated with that type to label the object. You can pass objects satisfying that interface to methods whose parameter types are that interface type, and you can return objects satisfying that interface from a method whose return type is that interface. The Counting in the previous paragraph was an example of the power of interfaces.

However, there are certain things that you cannot do with an interface.

Of course, when we're manipulating that Counting object, we don't know anything about how it works inside. We don't know, for example, whether it has a touchdown part and a field goal part, or is represented in decimal or in binary, or is likely to keep going up while we're thinking about it (since players might keep scoring). To figure this out, we'd need to know more than just the interface — the contract — that it satisfies; we'd need to know how it is implemented.

Interfaces are about contracts, promises. They don't, for example, tell you how to create objects that satisfy those promises. In the next several chapters, we'll learn about building implementations that satisfy these promises and about creating brand new objects that meet these specifications. To do that will require additional machinery beyond the contract/promise of an interface.

Style Sidebar

Interface Documentation

An interface should be properly documented, typically using a multi−line or Javadoc comment immediately preceding its declaration.

Documentation for an interface should include the following information:

- What kind of thing does this interface represent? Why would you want to use an object of this kind? What could it do for you? What could you do with it?
- What kinds of assumptions or conditions does this kind of object need to do its job? Are there any special objects that it might need to have around or to work with?
- What services does this kind of object provide, and how do you use them? These questions are typically answered by the individual methods, but a brief overview of what methods the interface provides is always useful. It is may also be useful for the interface to document which method(s) to use when, especially when multiple similar methods exist.

The interface's documentation should make it easy for a potential user to find the method(s) the user wants. It should also make it possible for someone seeking to implement this interface to determine whether the user has met the intent as well as the formal specification of the interface. If I am building a stopwatch, do I want to subscribe to the Clock interface?

Remember that an interface declaration is largely about *what*, not *how*. It specifies contracts and promises, not mechanism.

Java provides additional support for some of these items in its Javadoc utilities. See the appendix on *Javadoc* for details.

Chapter Summary

- An ***interface*** is a contract that a particular kind of object promises to keep.
- Java interfaces are Java types.
- Every (public) interface must be declared in a Java file with the same name as the interface.
- Java interfaces contain method signatures.
- A ***method signature*** specifies a method's name, parameter types, and return type. It does not say anything about how the method actually works.
- A method signature is also called an `abstract` method.
- One interface may have multiple methods with the same name, as long as they have different ordered lists of parameter types. Method name plus ordered parameter type list is called the method's ***footprint***. Having two methods with the same name but different footprints is called ***overloading***.
- An interface does not contain enough information to create a new object, though it can be used as a type for an existing object (that implements the interface's promises).
- Many important properties of a method specification or interface are not specified by the method or interface declaration. Good documentation describes these additional assumptions.

Exercises

See the text for questions marked **Question:**. Also:

1. StringTransformer has a *transform* method. Declare an interface, Transformer, that contains this single method specification, so that StringTransformer might be an implementation of this interface.

2. A Clock is an object that needs a method to read the time (say, *getTime*) and one to set the time (say *setTime*). Assuming that you have a type Time already, write the interface for a Clock.

3. Extend the interface of Clock (from the previous exercise) to include a *setAlarm* method that should specify the Time at which the alarm should go off.

4. Extend the Clock interface further so that there is a second *setAlarm* method that takes a Time and a boolean specifying whether the alarm should be turned on.

5. Write the interface AlarmedCounting.

6. Consider the following interface:

```
interface Game {

    /* returns the Counting that keeps track
       of the team's score */
    Counting getScoreCounter(Team team);

    /* returns the Counting that keeps track
       of how many fouls each player has committed */
    Counting getFoulCounter(Team team, int playerNumber);

    /* returns the AlarmedCounting that keeps track
       of how much time has passed in the period so far */
    AlarmedCounting getTimeCounter();

    /* returns the length of a period */
    int getPeriodLength();
}
```

Assume that *theWorldCup* is a particular Game, according to this interface.

a. Write a type declaration for the name *theWorldCup*. Don't worry about where its value comes from.

b. Write a type declaration suitable for holding the result of **`theWorldCup.getTimeCounter()`**.

c. Write an expression that returns the object that counts the fouls of player 5 on Team *philadelphiaFlyers* .

d. Write an expression that returns the current score of Team *philadelphiaFlyers* in *theWorldCup*.

e. Write a method invocation that sets up *theWorldCup* (and its internal representation) so that it will print "Period over!" when the elapsed time reaches the length of the period.

Chapter 5

Expressions: Doing Things with Things

Chapter Overview

- How do I use the Things I have to get new (or other) Things?

This chapter and the next introduce the mechanics of executable code, the building blocks for individual sequences of instruction−following. The previous chapter's *Things* each come with a *Type*, which specifies how that Thing can interact. An ***expression*** is a piece of code that can be evaluated to yield a value and a type.

Simple expressions include ***literals*** — Things that Java literally understands as you write them — and ***names***, which stand in for the Things to which they refer. More complex expressions are formed by combining other Things according to their *types*, or promised interactions.

To understand a complex expression, you must understand its parts (a basic form of "what goes inside") and how they are combined (a basic "how they interact"). Sometimes, you have to understand this without knowing all of the details of what's inside.

Sidebars in this chapter cover details of various Java operators, including *casts* and *coercion rules*. In addition, supplementary reference charts are provided outlining the syntax and semantics of Java expressions.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Objectives of this Chapter

1. To understand that an *expression* is a piece of Java code with a *type* and a *value*.
2. To become familiar with the *rules of evaluation* for basic Java expressions.
3. To learn how to understand complex expressions as combinations of simpler expressions.
4. To learn how to understand complex expressions as combinations
5. To be able to evaluate both simple and complex expressions.

5.1 Simple Expressions

An ***expression*** is the simplest piece of Java code. An expression is a Thing, so it has both a ***value*** and a ***type***. An ***instruction−follower*** — an execution of Java code — ***evaluates*** an expression to obtain its value, which will always be of the expression's type. There are many kinds of expressions, and each has its own rules of evaluation that determine what it means for an instruction−follower to evaluate that expression. Legitimate Java expressions include the following:

```
2 + 2
"Hi, there"
this.out.writeOutput(this.in.readInput())
```

The last of these is an expression whose evaluation involves inter−object (and inter−entity) communication.

5.1.1 Literals

The very simplest Java expression is a ***literal***: an expression whose value is interpreted literally, such as:

```
25
32e−65
"How about that?"
```

Java literals include the various kinds of numbers, characters, Strings, and booleans. For a more complete enumeration of literal expressions and rules regarding their ***syntax*** (i.e., how you write them), see the sidebar on *Java Primitive Types* in Chapter 3, *Things,*

Types, and Names.

Every expression has a value and a type, obtained by evaluating the expression. The value of a literal is its *prima facie* value, i.e., what it appears to be. The type of an expression is the type of its value. Integer literals are always of type `int` unless an explicit type suffix (*l*, *s*, or *b*) is included in the literal. Non−integral numeric literals are always of type `double` unless explicitly specified to be of type `float` (using the *f* suffix).

5.1.2 Names

Names are also Java expressions. A name is a legitimate expression only once it has been declared, i.e., within its ***scope***.

[Footnote: The area of text within which a name is legal is called its ***scope***. The scope of a ***variable*** — a name with no special properties beyond being a name — begins at its declaration and extends to the end of the enclosing block. (See the discussion of *blocks* in Chapter 6, *Statements and Rules*.) Later, we will see three other kinds of names: ***classes***, ***fields*** and ***parameters***. Class names have scope throughout a program or package; they may be used anywhere. Field names have scope anywhere in their enclosing class, including textually prior to their declaration. Parameter names have scope throughout their method bodies only.]

The value of that name is the value currently associated with it — i.e., the setting of the dial if it is a dial name, or labeled by it if it is a label name. The type of a name expression is always the type associated with that name at the time of its declaration.

[Footnote: Note that the type of a name expression is the declared type of the name rather than the type of the value associated with the name. That is, even where there is disagreement between the declared type of a name and its value, the type of a name expression is always its declared type.]

For example, if we are within the scope of a declaration that says

```
int myFavoriteNumber = 4;
```

and nothing has occurred to change the value associated with (set by the dial called) *myFavoriteNumber*, then the value of the expression

```
myFavoriteNumber
```

is *4* and its type is `int`. That is, the `int` *4* is the result of evaluating *myFavoriteNumber*.

5.2 Method Invocation

Method invocation is the primary way in which one object asks another to do something. It is the primary basis for inter−entity communication and interaction, because it is the main way in which objects talk to one another.

We have seen in previous chapters that objects are able to perform certain services. These service requests are called ***methods***, and asking an object to do something is called ***method invocation***. In Java, a method invocation involves:

- An expression whose value is the object to whom the request is directed, followed by
- A period (or "dot"), followed by
- The name of the method to be invoked, followed by
- Parentheses, within which any information needed by the method request must be supplied.

An example method invocation might be:

```
"a test string".toUpperCase()
```

This example consists of a String literal expression (**`"a test string"`**) and a request to that object to perform its *toUpperCase* method. A String's *toUpperCase* method doesn't require any additional information, so the parentheses are empty. (They can't be omitted, though!) The value of a String's *toUpperCase* method is a new String that resembles the old one, but contains no lower−case letters. So the value of this expression is the same as the value of the literal expression **`"A TEST STRING"`**.

Another example of method invocation is:

```
Console.println("Hello")
```

This asks the object named by the name expression *Console* to print the line supplied to it. It requires that a String — the line to be printed — be supplied inside the parentheses. This "necessary information" is called an ***argument*** to the method.

What is the value of this method invocation expression?
`Console.println("Hello")` is a method invocation whose primary use, like that of assignment, is for its side effect, not its value. We use this method to make something appear on the user's screen. Good style dictates that we wouldn't use this expression inside any other expression. It turns out that many methods have no real return values, so (as we saw in the previous chapter) there's a special Java type for use on just such

occasions. This type is called `void`. It is used only for method return types, and it means that the method doesn't return anything.

The evaluation rule for a method invocation expression is as follows:

1. Evaluate the object expression to determine whose method is to be invoked.

2. Evaluate any argument subexpressions.

3. Evaluate the method invocation by asking the object to perform the method using the information provided as arguments.

4. The value of the expression is the value returned by the method invocation. The type of the method invocation expression is the declared return type of the method invoked.

In order for step 3 to work, the object must know how to perform the method, i.e., it must have instructions that can be followed in order to produce the return value needed in step 4. We have already seen how an interface can describe an object's commitment to provide such behavior. We will see in the next chapters how this may be accomplished in detail.

From the perspective of the method invoker, however, the transition from step 3 to step 4 happens by magic (or by the good graces of the object whose method is invoked). The object offers the service of providing a particular method requiring certain arguments and returning a value of a particular type. For example, if we look at the documentation (or code) for String, we will see that it has a *toUpperCase* method that requires no arguments and returns something of type String. The *println* method of *Console* requires a String as an argument, and *println*'s return type is `void`. We will learn more about the methods that objects provide in Chapter 7, *Building New Things: Classes and Objects*, and Chapter 8, *Designing with Objects*.

5.3 Combining Expressions

Since expressions are things — with types and values — expressions can be combined to build more complicated expressions. For example, the expression

```
"serendipitous".toUpperCase()
```

has the type String and the same value as the literal **"SERENDIPITOUS"**. That is, you can use it anywhere that you could use the expression **"SERENDIPITOUS"**. So, for example, you could get an adverbial form of this adjective by using

```
"serendipitous".toUpperCase() + "LY"
```

producing **"SERENDIPITOUSLY"**, or you could extract the word **"REND"** using

```
"serendipitous".toUpperCase().substring(2, 6)
```

In general, since every expression has a type, you can use the expression wherever a value of that type would be appropriate. The exception to this rule about reuse of expressions is that some expressions are ***constant*** — their value is fixed — while other expressions are not. Some contexts require a constant expression. In these cases, you cannot use a non−constant expression of the same type.

For example,

```
"to" + "get" + "her"
```

is a constant expression, but

```
str + "ether"
```

is (in general) not, even if *str* happens to have the value **`"tog"`**.

[Footnote: The expression **`str + "ether"`** would be constant if *str* were declared `final`, though. Names declared to be `final` cannot be assigned new values.]

There are a few places where Java requires a constant value. These will be noted when they arise.

The evaluation rule for a compound expression is essentially the same as the evaluation rules for the expressions that make it up: Evaluate the subexpressions that make up this expression, then combine the values of these subexpressions according to the evaluation rule for this expression.

For example, when we evaluate

```
"serendipitous".toUpperCase()
```

we are actually evaluating the simpler (literal) expression **`"serendipitous"`**, then evaluating the method invocation expression involving **`"serendipitous"`**'s *toUpperCase* method. Similarly, **`str + "ether"`** evaluates the (name) expression *str* and the (literal) expression **`"ether"`**, and then combine these values using the rules for + expressions, detailed below. In this case, *str* and **`"ether"`** are subexpressions of **`str + "ether"`**.

There are two additional details: 1) Evaluating the subexpressions may itself involve several evaluations, depending on how complex these expressions are, and 2) it may not always be clear which operation should be performed first. (See Section 5.7, *Parenthetical Expressions and Precedence*.)

Method invocation, like other expressions, can be used to form increasingly complex expressions. For example, we can combine two method invocations we used above to cause the value of **"A TEST STRING"** to appear on the user's screen:

```
Console.println("a test string".toUpperCase())
```

In this case, the value of the *toUpperCase* invocation is used as an argument to *println*. We can also cascade other kinds of expressions, such as

```
"This is " + "a test string".toUpperCase()
```

or

```
Console.readln().toUpperCase()
```

5.4 Assignments and Side−Effecting Expressions

Another kind of operator is ***assignment***. We have already seen some simple assignments — including some that were mixed with declarations and buried inside definitions. An assignment is actually a kind of expression. Its first operand — the expression on the *left−hand side* — must be a name or another expression that can refer to a dial or a label. **In this context, and in this context *only*, the name expression refers to the dial or label, *not* to the particular value currently associated with the name.**

Like all expressions, every assignment has a type and returns a value. The type of an assignment is the type of its left−hand side. The value of an assignment expression is the value assigned to the left−hand side. For example, the type of the expression

```
myNumber = 4563129
```

is `int`, because the type of **4563129** is `int`, and the value is **4563129** for the same reason.

Note that we must have declared *myNumber* before we get to this expression; and that this expression is legitimate only if *myNumber* has type `int`, `long`, `float`, or `double`. Note, also, that if *myNumber* were already declared, we wouldn't want to declare it again. Every time that you declare a name, it creates a brand new dial or label with that name.

Although assignments are expressions in Java, they are not generally used for the resulting value. Instead, an assignment statement is generally used because it will cause the dial or label on its left−hand side to be associated with a new value. This effect is not a part of the value of the expression; instead, it happens "on the side" and is called a ***side effect***. Assignment statements are among the most common expressions used for their side effects, but we will see several other expressions with important side effects in the remainder of this chapter.

Style Sidebar

Don't Embed Side−Effecting Expressions

When you use a side−effecting expression, it is best if this expression is not a subexpression of any other expression. So, for example, while assignments — as expressions — *can* be used inside other expressions, it is generally considered bad style to do so. Embedding side−effecting expressions inside other expressions can make the logic of your code very difficult to follow. Side effects are also important and often difficult to catch. By highlighting the side−effecting expression by making it the outermost expression, you are increasing the likelihood that it will be read and understood.

5.5 Other Expressions That Use Objects

We have already seen method invocation, perhaps the most common object expression. In this section, we cover three additional expressions that use objects: *field access*, *instance creation*, and *type membership*. Each of these kinds of expressions will be discussed further when we explore how objects are actually created, beginning in Chapter 7, *Building New Things: Classes and Objects*.

5.5.1 Fields

In addition to methods, objects sometimes have ***fields***: data members that behave as names. That is, fields are either dials or labels. Like methods, fields are also accessed using the dot syntax, but without following parentheses. A ***field access expression*** is essentially a name expression, though a more complex one than the simple names described above. The value of a field access expression is, as for a simple name, the value associated with the dial or label. So, for example, *Math.PI* is a `double` dial, belonging to an object called *Math*, containing a value approximating a real number whose most significant digits are *3.14159*.

We can use field invocations in compound expressions, too. If *myWindow* is a Window with a *getSize* method that returns a Dimension, then

```
myWindow.getSize().height
```

first asks *myWindow* to perform its *getSize* method, resulting in a particular Dimension object, then asks the Dimension object for its *height* field. This compound expression is the same as first creating a name for the Dimension and assigning it the result of the method invocation:

```
Dimension mySize = myWindow.getSize();
```

and then asking the newly named Dimension object called *mySize* for its *height* field.

Because field access expressions are actually name expressions, they also have special behavior in the specific context of the target of an assignment statement. That is, you can assign to a field access expression just as you would to a simple name, and the field access expression behaves like the dial or label to which it refers. For example, if *height* is an `int` dial owned by *mySize*, the expression

```
mySize.height = mySize.height / 2
```

halves the value contained in the *height* dial of *mySize*, which might shrink *mySize* vertically by half.

5.5.2 Instance Creation

A second object−related expression is the ***instance−creation expression***, used with a ***class name*** to create a new object. The details of this expression type are covered in Chapter 7, *Building New Things: Classes and Objects*; for now it is enough to recognize it. An instance−creation expression has three parts: the keyword `new`, the class name, and a (possibly empty) list of arguments, enclosed in parentheses. This description of how to write an expression is called its ***syntax*** , and we can abbreviate the syntax of the instance−creation expression as:

Syntax of an Instance−Creation Expression
`new` *`ClassName`* `(`*`argumentList`*`)`

The words in italics — *ClassName* and *argumentList* — are placeholders to indicate that you need to supply the details. The rest of the expression — `new` and the parentheses — are to be taken literally. For example,

```
new File ("myData")
```

creates a new File object with external (outside of Java) name *myData*.

Like all other expressions, an instance−creation expression has a type — *ClassName*, the kind of object created, in this case File — and a value — the new object created. The instance−creation expression is typically used inside an assignment or method invocation.

The rules of ***evaluation for instance−creation expressions*** are similar to the rules of evaluation for method invocation. The return value is always a new instance of the type (or class) whose instance−creation expression is invoked (in this case, File). The return

type is always the type whose instance creation is invoked. Instance creation is a side−effecting expression (since it creates a new object).

5.5.3 Type Membership

There is one last operator that is useable only with objects. This is an operator called `instanceof`, which checks whether an object has (or can have) a certain type. It takes two operands:

Syntax of an `instanceof` Expression
anObjectExpression instanceof *ObjectTypeName*

The first operand, which precedes the keyword `instanceof`, can be any expression whose value is of any object (non−primitive) type. The second operand, which follows the keyword `instanceof`, must be the name of an object type. As we shall see in the next few chapters, this name may be the name of any class or any interface.

The `instanceof` operator is used to determine whether it is appropriate to treat its first operand according to the rules of the type named by its second operand. (For example, is it appropriate to "cast" the object to this type, as described in the next section?) The value of an `instanceof` expression is a `boolean`: `true` if it is appropriate to treat the object according to this type, `false` otherwise. So, for example,

```
"a String" instanceof String
```

has the value `true` (because **`"a String"`** is a (literal) instance of the type String), while

```
new Object() instanceof String
```

has the value `false` (because the new Object created by the instance−creation expression **`new Object()`** is not a String).

5.6 Complex Expressions on Primitive Types: Operations

Perhaps the most common kind of expression on primitive types is made up of two expressions combined with an ***operator***. Java operators are described in the sidebar on Java Operators. They include most of the common ***arithmetic operators*** as well as facilities for ***comparisons***, ***logical operations***, and other useful functions. Of special note is + for ***String concatenation*** (as well as for ordinary addition).

Each operation takes arguments of specified types and produces a result with a particular value and type. For example, if *x* and *y* are both of type `int`, so is **`x + y`**. The + ***operator*** can be used to combine any two numeric types. The two things combined with the operator are called the ***operands***. In the expression **`x + y`**, the + is the operator and *x* and *y* are the operands. Some operators take two operands. These are called ***binary operators***. Other operators take only one operand; these are the ***unary operators***. One operator — `?:` — takes three operands.

Java Operators

Java operators include:

```
+      -      *      /      |      &      ^      %      <<     >>     >>>
+=     -=     *=     /=     |=     &=     ^=     %=     <<=    >>=    >>>=
<      >      <=     >=     ==     !=
!      &&     ||
++     --
=      ?:
```

The ***arithmetic operators*** and ***bitwise−logical operators*** in the first row are, respectively, addition, subtraction, multiplication, division, bitwise or, bitwise and, bitwise negation, modulus, left−shift, sign−extended right−shift, and zero−extended right−shift. The + operator is also used for String concatenation when at least one of its arguments is a String. The − operator can also be used as unary (one−argument) negation.

The operators in the second row are ***operator−assignment operators*** that combine their correlate in the first row with an assignment operation. Thus

```
x += 2
```

has the same effect as

```
x = x + 2
```

The difference is that the left−hand side of the combined operator is evaluated only once. The value of an operator−assignment expression is the new value of the left−hand side; the type is the type of the left−hand side. All assignment expressions modify the name that is their left−hand side.

The third row above lists the six ***comparison operators***, each of which returns a `boolean`. The final comparison is not−equal.

The fourth row lists the ***logical operators***: ***logical negation***, ***logical conjunction***

(and), and ***logical disjunction (or)***. Each of these takes `boolean` arguments — one in the case of negation, two in the case of conjunction and disjunction — and returns a `boolean`.

The operators in the fifth row are ***autoincrement*** and ***autodecrement***. These can be used as either prefix or postfix operators. Both **`++x`** and **`x++`** modify *x*, leaving it incremented. However, **`++x`** returns the incremented value of *x*, while **`x++`** returns the unincremented value. The `--` operator works similarly.

The final two operators are simple assignment (which works like the compound assignments, above) and the ternary (three–operand) ***expression conditional***:

```
x > y ? a : b
```

evaluates to *a* if **`x > y`** , and to *b* otherwise. (You can use any `boolean`–valued expression where I used **`x > y`** , and any expressions where I used *a* and *b* .)

5.6.1 Arithmetic Operation Expressions

The operator + is an example of a kind of operator called an ***arithmetic operator***. The rules for evaluation of the binary arithmetic operators `+`, `-`, `*`, `/`, and `%` are simple: compute the appropriate mathematical function (addition, subtraction, multiplication, division, and modulus, respectively), preserving the types of the operands. As explained in the sidebar on *Arithmetic Expressions*, an expression of the form

```
type operator type
```

has type *type* for all of the basic arithmetic operations on most of the primitive types. That is, for these arithmetic operators, if the types of the two operands are the same, the result — the value of the complete expression — will generally also be of that type. For example, the expressions

```
3 + 7
2.0 * 5.6
5 / 2
```

evaluate to the `int` 10, the `double` 11.2, and — perhaps surprisingly — the `int` 2 (not 2.5 or 2.0), respectively.

Sometimes, an operator needs to treat one of its operands as though it were of a different type. For example, if you try to add `7.4` (a `double`) and `3` (an `int`) Java will automatically treat the `int 3` as though it were the equivalent double, `3.0`. This way, Java can add the two numbers using rules for adding two numbers of the same type. This kind of treating numbers — or other things — as though they had different type is called

coercion. Coercion does not actually change the thing, it simply provides a different version (with a different type). For dial types, this version is essentially a copy. For label types, it is another "view" of the same object. Coercion is described more fully in the sidebar on *Coercion and Casting*.

Other arithmetic operators work in much the same way as the + operator. Additional information on arithmetic expressions is summarized in the sidebar *Arithmetic Expressions*. Note in particular that `/` (the division operator) obeys the same *Type−Operator−Type is Type* rule. This means that **`7 / 2`** has type `int` (and the value **`3`**). If you want a more precise answer — **`3.5`** — you can make sure that at least one operand is a floating point number: **`7.0 / 2`** has type `double`, as does **`7 / 2.0`** and (best style) **`7.0 / 2.0`**.

In addition to the ***binary (two−argument) arithmetic operators*** described above, Java includes a ***unary minus operator*** that takes one argument and negates it. So **`-5`** is a (literal) `int`, while **`- 5`** is an arithmetic expression that has value **`-5`** and type `int`. (Subtle, no?)

Arithmetic Expressions

Arithmetic expressions include the binary operators for addition (+), subtraction (−), multiplication (*), division (/), and the modulus or remainder operation (%). In addition, there are two unary arithmetic operators, + and −.

Arithmetic operations work only with values of type `int`, `long`, `float`, or `double`. When a (unary or binary) arithmetic expression is invoked with a value of type `short`, `byte`, or `char`, Java automatically widens that operand to `int` (or to a wider type if the other operand so requires). For further details on ***widening***, see the sidebar on *Coercion and Casting*.

When the operands of a binary arithmetic expression are of the same type, the complete expression also has that type, except that no binary arithmetic expression has type `short`, `byte`, or `char`. This is because operands of these types are automatically widened.

When the operands are of different types, Java automatically widens one to the other.

The values of the expressions involving the binary operators +, −, *, and / are the sum, difference, product, and quotient of their (possibly widened) operands, respectively.

The value of **`x % y`** is the (appropriately widened) remainder when *x* is divided by *y*.

The value of a unary − expression is the additive inverse of its (possibly widened) operand; a unary + expression has the value of its (possibly widened) operand.

5.6.2 Explicit Cast Expressions

If the numbers you wish to divide — or otherwise combine — are not literals, you can still change their types using an ***explicit cast expression*** (as described in the sidebar on *Coercion and Casting*). Like coercion, this gives you a view of the thing cast as a different type. It is accomplished by putting the name of the type that you wish the thing to have in parentheses before the (expression representing the) thing. For example, if *myInt* is an `int`-sized dial showing the value **`3`**, then

```
(long) myInt
```

is a view of **`3`** as a `long` and

```
(double) myInt
```

is an expression with the same type and value as the literal expression `3.0`. Throughout this, *myInt* itself remains an `int`-sized dial showing the value `3`.

Evaluating a cast expression yields the value of the cast operand (in this case, *myInt*), but with the type of the explicit cast (in the first example above, `long`). A cast expression does not alter its operand in any way; it simply yields a new view of an existing value with a different type. Some casts are straightforward and appropriate; some risk losing information; and most are simply not allowed. For example, in Java you cannot cast an `int` to `boolean`. Casts are also allowed from one object type to another under certain circumstances. See the sidebar on *Coercion and Casting* for further details.

Coercion and Casting

Sometimes things don't have the types we might wish. ***Coercion*** is the process of viewing a thing as though it had a different type. Coercion does not change the thing itself; it merely provides a different view.

Java only makes certain automatic — implicit — coercions. For example, Java knows how to make `byte` into `short`; `short` into `int`; `int` into `long`; `long` into `float`; and `float` into `double`. This works because each type spans at least the magnitude range of the ones appearing before it in the list. (A few of these coercions — such as `long` to `float` — may lose precision.) These coercions — which are, in general, information-preserving — are called ***widening***. We will see in Chapter 7, *Building New Things: Objects and Classes*, that there are also widening coercions on label (reference) types.

Coercions in the opposite direction are called ***narrowing***. Java does not generally perform narrowing coercions automatically. For example, Java cannot automatically convert an arbitrary `int` to a `short`, because the `int` might contain too much information to fit into a `short`. The number 60000 is a perfectly legitimate value for an `int`, but not for a `short`. There is no mapping from `int`s to `short`s that accurately captures the magnitude information in each possible `int`. A coercion of this kind — such as `int` to `short` — which may not preserve all of the information in the original object, is called ***lossy***.

[Footnote: There is one instance in which Java performs a narrowing but non-lossy coercion automatically. This is in the case of a sufficiently small `int` constant assigned to a narrower integer type. This allows literals — which would otherwise have type `int` — to be assigned to names with `byte` and `short` type, e.g.:

```
short smallNumber = 32;
```

]

Sometimes, you need to change the type of an object when Java will not do so automatically. This is accomplished by means of an ***explicit cast expression***. The syntax of a cast expression is:

Syntax of a *cast* Expression
(*type-name*) *expression-to-be-cast*

For example, if *myInt* is a name of type `int` with value 7, for example by

```
int myInt = 7;
```

then

```
(long) myInt
```

is an expression with type `long` and value 7. (Note that *myInt* still has type `int`. Casting, like implicit coercion, does not actually modify the castee.)

Explicit coercion allows both widening and narrowing coercions: you can cast an `int` to `long`, as in the example above, or to `short` — a cast that may lose information. A typical example of a lossy cast is to convert a `double` to an `int`. For example, if *x* and *y* are `doubles`,

```
(int) ((x + y) / 2.0)
```

evaluates to their (truncated) integer average. Certain casts may be illegal and will cause (compile-time or run-time) errors or exceptions.

5.6.3 Comparator Expressions

Not all operators are arithmetic. There is a set of boolean-yielding operators, sometimes called ***comparators***, that operate on numeric types. These include <, <=, ==, etc. (See the sidebar on *Java Operators* for a complete list.) These take two numbers, coerce appropriately, and then return a `boolean` indicating whether the relationship holds of the two numbers in the order specified. For example,

```
6 > 3.0
```

is `true`, but

```
5 <= 3
```

is `false`.

Beware: == tests for equality; = is the assignment operator.

Equality testing — the operators `==` and `!=` — are not restricted to numeric types. For any type, these operators combine two expressions of the same type, returning `true` only if both operands are the same. When are two operands the same?

- For primitive types, values are the same whenever they "look" the same, i.e., when their values are indistinguishable. For example, two "different" copies of the (int−sized) number 3 are, for purposes of testing for equality, the same.

  [Footnote: However, this does not extend to 3 and 3.0 and 3.0f, each of which is a different thing. This is because each of these has a different type. Attempting to compare two operands of different type yields a compile−time error.]

- For object types, values are the same exactly when the two expressions refer to the *same object*. It is not sufficient for two objects to look alike (as in the case of identical twins); they must actually be the same object, so that modifications to one will necessarily be reflected in the other.

Consider, for example, identical twins X and Y. Although they may look exactly the same, they are still two different people. If one gets a haircut, the other's hair doesn't automatically get shorter. If one takes a bath, the other doesn't get clean. Thus they are different: **`X == Y`** is `false`.

The `int` 3, on the other hand, has no internal structure that can be changed (the way that one twin's hair can be cut). If you change 3, you don't have 3 any more. If dial name A and dial name B each are of type `int` and each have 3 as their value, then **`X == Y`** is `true`.

Evaluating one of these expressions is much like evaluating an arithmetic expression. The values of the operands are compared using a rule specific to the operator — such as `>` or `<=` — and the resulting `boolean` value is the value of the expression.

5.6.4 Logical Operator Expressions

Another set of operators combines booleans directly. These include `&&` (***conjunction***, or "***and***") and `||` (***disjunction***, or "***or***"). For example, the expression

```
true || false
```

is `true`. While this is not very interesting by itself, these ***boolean operators*** can be used with names (of type `boolean`, of course) or in complex expressions to great effect. For example,

```
rainy || snowy
```

might be a reasonable way to express bad weather; it will (presumably) have the value `true` exactly when it is precipitating. There is also a unary boolean ***negation operator*** denoted `!` (exclamation point). The Java fragment

```
! (rainy || snowy || overcast)
```

might be a good expression for sunshine.

The rule for evaluating negation is simply to invert the boolean value of its operand. The rules for evaluating conjunction and disjunction are a bit more complex. First, the left operand is evaluated. If the value of the expression can be determined at this point (i.e., if the first operand to a conjunction is false or the first operand of a disjunction is true), evaluation terminates with this value. Otherwise, the second operand is evaluated and the resulting value computed. The type of each of these expressions is `boolean`.

These odd–seeming rules are actually quite useful. You can exploit them to insert tests. For example, you might want to compute whether

```
(x / y) > z
```

but it might be the case that *y* is 0. By testing whether

```
(y == 0) || ((x / y) > z)
```

you can eliminate the potential divide–by–zero error. (If *y* is 0, the first operand to the disjunction — **`(y == 0)`** — will be true, so evaluation will stop and the value of the whole will be `true`.) A comparable formula can be written to return `false` if either *y* is 0 or **`(x / y) > z`**.

5.7 Parenthetical Expressions and Precedence

A ***parenthetical expression*** is simply an expression wrapped in a pair of parentheses. The value of a parenthetical expression is the value of its content expression, i.e., the value of the expression between the opening parenthesis and the closing parenthesis. The type of a parenthetical expression is the same as the type of the expression between the parentheses.

Parenthetical expressions are extremely useful when combining expressions. For example, suppose that the name *x* has the value **`6`** and consider the expression:

```
"I have " + x + 3 + " monkeys"
```

The human writing this expression might have meant

```
"I have " + (x + 3) + " monkeys"
```

(which evaluates to *"I have 9 monkeys"*), but in Java this expression is equivalent to

```
(("I have " + x) + 3) + " monkeys"
```

(which evaluates to *"I have 63 monkeys"*). Isolating **`x + 3`** as a separate expression makes the + in **`x + 3`** behave like addition, not String concatenation.

Note that, in giving the evaluation rules for expressions, white space doesn't matter:

```
x >= 2 + 3
```

is identical to

```
x >= 2      + 3
```

but punctuation does matter. For example,

```
2 + 3 * 2
```

doesn't have the same value as **`5 * 2`** —

```
2 + 3 * 2
```

is **`8`**. We can use parentheses to fix this, though:

```
(2 + 3) * 2
```

is **`10`** again. In this case, parentheses change the order of evaluation of subexpressions (or, equivalently, how the expression is divided into subexpressions.) In the case of

```
2 + 3 * 2
```

if you evaluate the + first, then the `*` , you get **`5 * 2`**, while if you evaluate the `*` first, you get **`2 + 6`**.

How do you know which way an expression will be evaluated? In these situations, where one order of operation would produce a different answer from another, we fall back on the rules of ***precedence*** of expression evaluation. In Java, just as in traditional mathematics, `*` and `/` take precedence over + and `–`, so

```
2 + 3 * 2
```

is really is 8. (Another way of saying this is that the `*` is more powerful than the `+`, so the `*` grabs the 3 and combines it with the 2 before the + has a chance to do anything. This is what we mean when we say that `*` has higher precedence than `+`: it claims its operands first.)

A full listing of the order of precedence in Java is included in the sidebar on <u>*Java Operator Precedence*</u>. Parentheses have higher precedence than anything else, so it is always a good idea to use parentheses liberally to punctuate your expressions. This makes it far easier for someone to read your code as well.

Java Operator Precedence

Expressions with multiple subexpressions are evaluated according to the rules of Java ***precedence***. The following chart gives the rules for order of evaluation of Java expressions, with the expression types listed higher having higher priority, i.e., being evaluated first.

Operators in the table below are grouped by equivalent precedence. Within these groups, order of evaluation of an expression is from left to right in that expression.

Since an expression cannot be evaluated until its subexpressions have been, precedence determines the extent of operands to each operator, i.e., what the operand subexpressions of an operator are.

<table>
<tr><td>++ -- + - ~ ! explicit cast</td></tr>
<tr><td>* / %</td></tr>
<tr><td>+ -</td></tr>
<tr><td><< >> >>></td></tr>
<tr><td>< <= > >= instanceof</td></tr>
<tr><td>== !=</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?:</td></tr>
<tr><td>= and all compound assignments</td></tr>
</table>

Other Assignment Operators

Compound Assignment

Java has several variants on the simple assignment statement. If we have already declared *total* as an `int`, we can say:

```
total = 6
```

or

```
total = total + 1
```

The second expression uses the fact that **`total + 1`** is an expression with type `int` and value one greater than **`total`** to form an expression whose second operand is an arithmetic expression. This last expression — adding to a name — is pretty common, and so it has a convenient shorthand:

```
total += 1
```

The ***+= operator*** is one of a class of ***compound assignment operators***. It works by computing the value of its first operand, then adding its second operand to that value and assigning the result to the name represented by the first operand. In other words, the expression above is exactly the same as saying:

```
total = total + 1
```

This kind of compound assignment can be used with any number — or other appropriate expression — as the second operand, of course. There are also other compound assignment operators in Java, including **`-=`** , **`*=`** , **`/=`** , and **`%=`** . Like the + operator, the += operator works for both numeric addition and String concatenation. Like their longhand forms — the simple assignment equivalents — these expressions have type and value of their left−hand side (after the assignment).

AutoIncrement and AutoDecrement

There is another family of side−effecting operators that are related to assignment. These operators are ***autoincrement*** and ***autodecrement***. The ***postfix*** autoincrement expression:

```
total++
```

is similar to **`total = total + 1`** (or **`total += 1`**), but it has the value of *total* **before** the assignment. The ***prefix*** autoincrement expression:

```
++total
```

also adds one to *total*, but has the value of *total* **after** the assignment. (Remember: ***`++variable`*** first increments, then produces a value; ***`variable++`*** produces the value first.) The two (prefix and postfix) autodecrement operators work similarly.

Chapter Summary

- Every ***expression*** has both a *type* and a *value*.
- ***Simple expressions*** include *literals* and *names*.
    - ♦ A literal has its apparent type and value.
    - ♦ A name has its declared type and assigned value.
- ***Operator expressions*** combine or produce modifications of simpler expressions.
    - ♦ ***Arithmetic operators*** compute mathematical functions; the type of an arithmetic operation expression is typically the wider of its operand types.
    - ♦ ***Logical operators*** compute binary logical functions; the type of a logical operation expression is `boolean`.
    - ♦ ***Explicit cast expressions*** have the type of the cast operation and the same value as the cast operand.
    - ♦ None of the above expressions actually modifies any of its operands. However, ***autoincrement***, ***autodecrement***, and the ***shift operators*** *do* modify their operands.
- ***Assignment expressions*** are generally used for their effects — modifying the value associated with a (dial or label) name — but, as expressions, also have type and value. The value of an assignment expression is the value assigned; the type is the type of the value assigned.
- Several kinds of expressions operate on objects:
    - ♦ A ***method invocation expression*** has the type and value returned by the method. Methods may be side−effecting.
    - ♦ A ***field access expression*** is like an ordinary name expression: its type is the field's declared type and its value is the field's current assigned value, except in the context of assignment expressions.
    - ♦ A ***constructor expression***'s value is a brand new object whose type is the type with which the constructor expression is invoked.

Exercises

1. In Java, every expression has a type. Assume that the following declarations apply:

```
int i, j, c;
double d;
short s;
long l;
float f;
boolean b;
```

For each expression below, if it is syntactically legal Java, indicate its type (*not its value*). If it is not syntactically valid, indicate why.

a. **6**
b. **24L**
c. **+3.5**
d. **3.5f**
e. **2e-16**
f. **-25b**
g. **i**
h. **i + 3**
i. **i + 3.0**
j. **i + s**
k. **l + d**
l. **f + s**
m. **i / 0**
n. **4 * 3.2**
o. **i = 0**
p. **i == 0**
q. **b = 0**
r. **b == 0**
s. **'c'**
t. **"An expression in double-quotes"**
u. **"An expression in double-quotes" + "another one"**
v. **"6" + 3**
w. **!b**
x. **!i**
y. **b || true**
z. **i += s**
aa. **s += i**
ab. **i += f**
ac. **l = i = s**
ad. **i = l += s**

ae. **`l++`**
af. **`(long) s`**
ag. **`s`**
ah. **`(short) l`**
ai. **`l`**

2. Give three examples of expressions with side effects.

3. What is the value of each of the following expressions? Which ones produce errors in evaluation? You may wish to consult the sidebar on *Java Operator Precedence* . Assume that the following definitions have already been made:

```
int i = 93;
boolean b = true;
```

a. **`2.0 + 3.5 * 7`**
b. **`("top " + "to " + "bottom").toUpperCase()`**
c. **`"the answer is " + 6 * 7`**
d. **`4 + 6 + " is " + 10`**
e. **`i > 0 && i < 100`**
f. **`b = i < 0`**
g. **`! (i == 0) && 100 / i`**

4. Give examples of each of the following. Throughout, assume that *x* and *b* are previously defined names for an `int` and `boolean` , respectively.

a. An expression whose type is `int` and whose value is more than *x*.

b. An expression whose type is `boolean` and whose value is `true` when *x* is between 5 and 15.

c. An expression whose type is `double` and whose value is half of *x*'s.

d. An expression whose type is `long` and whose value is the remainder when *x* is divided by 7.

e. An expression whose type is `boolean` and whose value is the opposite of *b*'s value.

f. An expression whose type is `boolean` and whose value is `true` exactly when *x* is evenly divisible by 5.

g. An expression whose type is String and whose value is read from the user's keyboard.

Chapter 6

Statements and Rules

Chapter Overview

- How do I tell the computer how to do something?

This chapter introduces statements, the simplest forms of complete executable instructions. Statements are fragments of Java code that have neither value nor type; instead, they have effects. Statements can be combined to form rules, or services that one object can provide to another. Statements and rules form the backbone of the peanut−butter and jelly model of programming.

Statements can be built out of expressions. However, unlike expressions, which have both type and value, statements are used for their effect — to get something done. Examples of this are asking a thing to do something or assigning a name to keep track of a value. In addition to declarations, assignments, and method invocation, this chapter introduces simple control flow statements. More advanced statement types are introduced later in the book.

The chapter ends with a discussion of methods, the rules implementing behavior. Method invocation provides the basis for virtually all inter−object interaction.

This chapter is supplemented by a reference chart on the syntax and semantics of Java statements.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Objectives of this Chapter

1. To appreciate the difference between evaluating an expression and executing a statement.

2. To be able to read and understand basic statements including assignments, method invocations, declarations, blocks, conditionals, and loops.

3. To learn how to combine statements to construct rules that implement method behavior.

6.1 Statements and Instruction−Followers

In the first chapter of this book, we saw that computations are made of communities of interacting entities. Each of these entities may be a community of smaller entities, until eventually an entity can be subdivided no more. At that point, an entity is a simple instruction−follower that provides behavior — often in the form of ongoing services — to the other members of its community. This chapter is about how those instructions work. Towards the end of the chapter, we will begin to see how instructions can be combined to form special sequences that articulate how service requests can be fulfilled.

In the previous chapter, we saw how to create Java expressions. An expression is a piece of Java code with a value and a type. The process of producing the value from an expression is called ***evaluating*** that expression. The purpose of evaluating an expression is generally to produce its value.

In contrast, statements are all about their side effects. A statement is a piece of executable Java code without either a type or a value. That is, a statement does something (changes something, produces some visible behavior, etc.). It has an effect. It does not have a value. A statement is ***executed*** (producing an effect), not evaluated (producing a value).

In order to evaluate an expression, you must evaluate its subexpressions, then use the evaluation rule for that kind of expression to produce an appropriate value of an appropriate type. If you understand the evaluation rules for each type of expression, you understand how expressions work.

Understanding how to execute a statement is similar. A statement is not defined by a type and a value (it doesn't have either!), but by its *effects* and by *what happens next*. That is, statements do things; they change the values associated with names. And statements can also cause you to skip around in the instructions that you are following. This is called

flow of control: what instruction to follow next. Some of these control flow statements involve ***conditions*** (*if* it's raining, do this) or ***loops*** (keep doing this *until* the light changes color). And many statements involve either subexpressions—which must be evaluated—or substatements—which must be executed in order to execute the superstatement.

6.2 Simple Statements

Perhaps the simplest kind of statement is one built directly out of an expression, such as

```
this.who = name;
```

or

```
Console.println("Hello");
```

Note the trailing semicolon following the ends of these expressions. It is this semicolon that converts these expressions into statements.

What kinds of expressions can be used to form statements? Only side−effecting expressions. Many expressions are useful solely because of the value that they compute. But a statement doesn't have a value; it has effects on state and control flow. So an expression whose primary purpose is the value it produces doesn't make a very good basis for a statement on its own.

[Footnote: These expressions may find use in other, more complex statements, though.]

In fact, it is not legal in Java to make an expression−semicolon statement out of a non−side−effecting expression. (For example, **`x + 3;`** is not a legal statement.)

However, some expressions do more than just produce values when they are evaluated. For example, an expression like **`x = 3`** has the value `3` (and the type `int`, assuming that *x* is an `int`). It also (and more importantly) has the effect of storing the value 3 in the dial named *x*. This effect (of evaluating the expression) is called a ***side effect***. All assignment expressions (including compound assignments) are side effecting. Autoincrement and autodecrement are also side−effecting expressions. Method invocation expressions are also side−effecting, although not every method invocation actually has a side effect. Instance creations — `new` expressions — are also side−effecting.

So, for example, a simple assignment statement can be made by adding a semicolon to the end of the assignment expression **`x = 3`**

```
x = 3;
```

The semicolon turns this into a statement. It no longer has a value or a type; it just does its work.

To execute an expression−semicolon statement, simply evaluate the expression. Of course, this expression may have complicated subexpressions that must be evaluated according to the rules described in the previous chapter. Since the expression is a side−effecting one, something will happen — an effect will be produced — during the evaluation.

After executing a side−effecting−expression−plus−semicolon statement, execution proceeds at the following statement.

6.3 Declarations and Definitions

We have also already seen declarations in Chapter 3, *Things, Types, and Names*. A declaration creates a new name that can be used to store (in the case of primitive types) or label (in the case of reference types) a value. A declaration follows the *Type−of−thing Name−of−thing* rule: It consists of a Java type followed by a Java name, then a semicolon. For example:

```
int i;
Object thing;
```

A declaration (or definition) statement creates a kind of name called a ***local variable***.

You can actually declare multiple names of a single type with one declaration statement. The syntax for this is *Type−of−thing Name−of−thing1*, *Name−of−thing2*, and so on, with commas between the names and a semicolon at the end:

```
int i, j, k;
Object thingOne, thingTwo;
```

The same type is associated to each of the comma−separated names, so the declarations above are identical to

```
int i;
int j;
int k;
```

and

```
Object thingOne;
Object thingTwo;
```

respectively.

Style Sidebar

Formatting Declaration Statements

Remember that Java doesn't care how much white space you leave between things, so there is no difference in meaning between putting the multiple declarations on one line or many. It is definitely easier to read on multiple lines, though, so the convention is to put each declaration on its own line.

When one declaration statement is used to declare many names, you can put the names on one line or on several. It's good style to indent all of the names on subsequent lines of a single declaration so that they line up with the first name declared:

```
Object thingWithALongName,
       anotherThingWithALongName;
```

This way, it's easy to see that *anotherThingWithALongName* is involved in the same declaration statement as *thingWithALongName*.

Although it is technically correct to mix declarations and definitions of a single type using the comma−separated multiple declaration notation, this is not good style. It is too easy to miss a definition among the declarations; mixing the two makes your code unnecessarily harder to read.

A declaration makes it legal to use the name to hold/label appropriately typed values. But the declaration, by itself, doesn't explicitly assign a value to the name. In fact, for the most generic kind of name—a local variable—it is illegal to use a name without first assigning it a value.

[Footnote: It is, however, legal to assign a label−name local variable the special non−value `null`. Assigning `null` to a name means that the name doesn't refer to anything. Not assigning forces the computer to guess. The rule is that you just can't leave the computer to guess.]

You can assign this value directly in the declaration (making it a definition), or you can assign it before the first time that you try to use the name's associated value.

A variant on a declaration statement is a ***definition***. A definition is a declaration statement with **`= expr`** between the *Name−of−thing* and the semicolon (or comma). This statement declares the name, but it also assigns it the value of *expr*. For example:

```
int    i = 2;
String who = "Pat";
double pi = 3.14159,
       ninetyDegrees = pi / 2;
```

Note that the final statement here assigns the value 1.570795 to the name *ninetyDegrees*. First 3.14159 is put into the dial named *pi*. Next, the expression **`pi / 2`** is evaluated: its value is the value inside the *pi* dial divided by 2. Finally, this value is assigned to (stored in) the (newly created) dial named *ninetyDegrees*.

It is legal to mix declarations and definitions in a single statement — assigning initial values to only some of the names — but this can make your code hard to read. It is usually better to use multiple statements in this case.

Executing a declaration statement creates a dial or label associated with the name declared. Executing a definition is the same as declaring a name, plus immediately afterwards executing an assignment statement. Note that this assignment is an expression and may have subexpressions, causing a significant amount of evaluation before execution is complete.

After executing a declaration or definition statement, execution proceeds at the immediately following statement.

6.4 Sequence Statements

You can also make a bigger statement out of a collection of statements. You do this by enclosing them in ***braces***:

```
{
    int i = 3;
    Console.println("i is " + i);
    int j = i + 1;
    j = i + 5;
}
```

This statement−made−of−statements is a ***block***, and it mostly serves to organize your code. Some other statements — such as `if`, described below — are often used together with blocks.

Any statement can be used at any point inside a block. In particular, declarations and definitions may appear anywhere in a block. This is useful as it allows you to declare a name immediately before you need it. Doing so makes it easier to read your code as the reader is less likely to have forgotten what you mean by that name.

Blocks also have implications for scoping of names: a variable has scope (its name can be used) from the point in the code where it is declared until the end of the first enclosing block.

[Footnote: Remember, not all names are variables. We will learn more about parameters and fields in subsequent chapters. Type names have scope everywhere that they are visible.]

So if we declare a name at the top of the block, it has scope for the whole block, as *i* does in the example above. But *j* is not declared until *after* the call to *println*, so the definition of *i* and the call to *println* are outside of *j*'s scope:

```
{
    int i = 3;                                  #
    Console.println("i is " + i);               #
    int j = i + 1;        #                     #  scope of i
    j = i + 5;            #  scope of j         #
}                         #                     #
```

This means, for example, that it would be illegal to use *j* in *i*'s definition:

```
{
    int i = j;    // illegal use of j outside its scope!
    Console.println("i is " + i);
    int j = i + 1;
    j = i + 5;
}
```

Beware: The scope of a local variable only persists until the end of the enclosing block. This means that a local variable must be declared at the same level as (or at a level enclosing) each of its uses.

```
{
    {
        // A variable declared here...
        String name;
    }
    // ...is invisible here, making this reference
    name = "Pat";
    // illegal!
}
```

The rules for executing a block statement are: execute each substatement in turn, from the top (beginning) of the block to the bottom (end) of the block.

After a block, execution continues at the next statement.

Style Sidebar

Formatting Blocks

The opening brace of a block should generally appear on its own line. If the block is part of a compound statement (such as an `if`), its opening brace can appear as the last character on a line. This is the style recommended in Sun's coding conventions at **www.java.sun.com/docs/codeconv**. However, some studies have found code using this convention harder for programmers to scan than code in which the open brace appears alone on a line.

Text within a block should always be indented (typically by two or four characters). This makes the left−hand margin of code in a block line up. The text — but not the braces — of an interior block is indented further; the original indent is resumed when the interior block is closed, i.e., after the closing brace.

The closing brace of a block should always begin its own line. If the closing brace completes the statement, as in a simple block, it should appear alone on that line.

```
// Some statements ...
{
    // Statements in a block all line up.
    {
        // Interior block statements
        // are indented further.
    }
    // Close brace exits the block
    // and restores earlier indent.
}
// ...and so on.
```

As mentioned above, the opening brace is often placed at the end of the portion of a statement whose body is the block, for example:

```
    if (blah) {
        // statements
    }
```

6.5 Flow of Control

So far, we have seen declarations, definitions, and a few executable statements made out of side−effecting expressions such as method invocation and assignment. You can write some interesting programs using only these constructs, but typical programs involve more complex structures. One of the most important features is the ability to control which code is executed when. This is called flow of control. These statements have execution rules that do not always cause the next statement to be executed in turn. Instead, a statement may be executed more than once or not at all.

6.5.1 Simple Conditionals

One of the simplest forms of control flow is ***conditional execution***. Conditional execution refers to a situation in which a block of code may or may not be executed, depending on the value of an expression. It is analogous to a set of instructions that says

> Step 1. If your gizmo is not already assembled, you must assemble it before going on to step 2. To assemble your gizmo, first...
>
> Step 2. Now that your gizmo is fully assembled, ...

In Java, conditional execution is most often and most generally embodied in the ***if statement***. For example:

```
if (theLight.isOn()) {
    theRoom.isLit = true;
}
```

Let's dissect this statement. It begins with the Java keyword `if`. After the `if` is a boolean expression that *must* be enclosed in parentheses. The closing parentheses are followed by a block statement.

[Footnote: There are other kinds of statements that can appear in place of this block, but in this book we will restrict ourselves to the cases in which the `if` body is a block.]

This block is sometimes called the `if` statement's ***body*** or the ***consequent***; the boolean expression is called the if statement's ***test*** or ***condition***.

Execution of the `if` statement proceeds as follows. First, the boolean condition expression is evaluated. If the value of this expression is true, the if's body block is executed. If the value of the boolean condition expression is false, the if's body block is skipped.

In either case, execution proceeds at the next statement following the if's body.

The `if` statement, as defined, is very useful when you want to do something or skip it. But often you want to do one of two things. We can express this using two `if` statements with inverse conditions:

```
if (theLight.isOn()) {
    theRoom.isLit = true;
}

if (! (theLight.isOn())) {
    theRoom.isLit = false;
}
```

This is poor code in three ways. The first is that it evaluates the same expression — **`theLight.isOn`** — twice, but the code would not work as we want if the values returned were different in the two evaluations. (Imagine that the light was off the first time you asked and on the second time. The value of **`theRoom.isLit`** would never get set!)

We could fix this problem by temporarily assigning this value to a boolean name, and then testing the name twice:

```
boolean itIsLight = theLight.isOn();

if (itIsLight) {
    theRoom.isLit = true;
}

if (! itIsLight) {
    theRoom.isLit = false;
}
```

But this makes a second problem with the code even more apparent. This code is testing a boolean expression (**`theLight.isOn()`** or **`itIsLight`**, depending on which version) in order to set another boolean expression. It would be cleaner just to write:

```
theRoom.isLit = theLight.isOn();
```

This statement is equivalent to the whole previous example (using *itIsLight*), and much easier to read. For more on this stylistic point, see the sidebar on Using Booleans.

Of course, we can write other code that's not subject to these two problems. For example, we could use this idea to write code to compute absolute value of a given `int`, *x*.

```
int absValue;

if (x > 0) {
    absValue = x;
}

if (x < 0) {
    absValue = - x;
}

if (x == 0) {
    absValue = 0;
}
```

This code has neither of the previous problems — *x* doesn't change, so we can test it repeatedly, and the value assigned is an int, not a boolean, so we can't write the shorter assignment statement. But this code doesn't make it clear that these are really three cases of the same test. There is a form of an `if` statement that allows us to make this clearer. It uses the Java keyword `else` to denote a situation in which we know that these conditions are mutually exclusive, i.e., at most one of them can hold.

So, for example, we could rewrite our light−tester (verbosely) as:

```
boolean itIsLight = theLight.isOn();

if (itIsLight) {
    theRoom.isLit = true;
} else {
    theRoom.isLit = false;
}
```

This still isn't as nice as the one−line version, but it gives us the opportunity to illustrate control flow in an ***if/else statement***. To execute an `if/else` statement:

1. Evaluate the boolean condition expression.

2. If the value of the condition is true, execute the `if` body block, then skip to the end of the entire `if/else` statement (i.e., to step 4).

3. Else (the value of the condition statement is false, so) execute the `else` body block. An `else` body is sometimes called an ***alternative***.

4. Execution continues at the following statement.

Since there might be more than two mutually exclusive conditions — as in the absolute value code — else is allowed to have its own condition. An else with a condition is like an if, except that you only execute that part of the statement if all previous conditions in this `if/else` statement have been false. An else with no condition is always executed if no previous condition in this `if/else` statement has been true.

```
if (x > 0) {
    absValue = x;
} else if (x < 0) {
    absValue = - x;
} else {
    absValue = 0;
}
```

Note that this is all one statement, not three as in the previous version. Exactly one of the assignment statements will be executed, no matter what the value of *x* at the beginning of the `if` statement.

Even now, this is not the most elegant absolute value code we could write; for example, the final case is redundant and could be folded into the first case using **`>=`** instead of **`>`**. It does, however, illustrate the syntax of ***cascaded `if` statements***. We will return to examine `if` statements, and other conditionals, in the chapter on Dispatch.

Style Sidebar

Using Booleans

There are only two `boolean` values, `true` and `false`. There can be lots of `boolean` labels, but each label is attached to either `true` or `false`; there is nothing else. This means that testing whether a `boolean` is the same as `true`, e.g.

```
(boolVal == true)
```

is redundant. You can just use *boolVal*, since it's either `true` or `false`. Similarly, you don't need to use an `if` statement to test a `boolean` if you're generating a `boolean` value. For example,

```
if (boolVal) {
    return true;
} else {
    return false;
}
```

is also redundant: just **`return boolVal;`**. The same thing applies if you're assigning to a variable instead of returning: **`otherBoolVal = boolVal;`** (or **`otherBoolVal = ! boolVal;`** if you want to reverse its sense).

6.5.2 Simple Loops

Another flow-of-control construct is ***while***. The `while` statement takes a condition and a block, just like the simple form of an `if` statement. Execution of a `while` statement first evaluates its boolean condition expression. If the condition is true, the `while` body block is executed. When execution of each statement in the body is complete, the `while`'s condition is checked again. Again, if the condition is true, the body is executed. This continues until the evaluation of the condition expression yields false; at this point, execution continues at the next statement *after* the `while` body.

There are several uses of a `while` loop. One is to continually test something until it becomes true:

```
int i = 1;

while (i < 100) {
    Console.println("I'm up to  " + i);
    i = i + 1;
}
```

This loop prints the numbers from 1 to 99. (Why doesn't it print 100?)

Another use is for a loop that keeps going essentially forever. (It will stop when something stops the program, but not before):

```
while (true) {
    myOutput.writeOutput(myInput.readInput());
}
```

This loop continually passes whatever input it gets to its output. Since the value of true doesn't change, this loop won't end until something nasty happens to it. Writing loops like this one — that go on essentially forever — is much easier than writing loops like the counting loop, above, because in the counting loop you have to keep track of what's true each time you go around the loop. For example, the value of i when you exit the loop above will always be one *more* than the last value printed.

Here's an even more tricky one:

```
while (x < 25) {
    x = x + 3;
    x = x - 2;
}
```

If *x*'s value is 20 when we reach the beginning of this loop, what will its value be when we exit? Remember that the test expression is only checked at the beginning of each pass through the loop, not in the middle.

There is another looping construct in Java, called ***do/while statement*** or just a ***do loop***. It is much like the `while` loop, except that the loop body is always executed once before the condition is tested:

```
int i = 1;

do {
    Console.println("I'm up to  " + i);
    i = i + 1;
} while (i < 100)
```

As with a `while` loop, once the loop exits, execution proceeds at the statement following the entire do statement.

6.6 Statements and Rules

Programs are not simply sequences of instructions to be executed. Instead, the instruction−followers executing these statements are embedded in a community of other instruction−followers. A program is a community of interacting entities providing ongoing behavior and services. In this section, we look at how those interactions too rely on statements.

When one Thing needs to communicate with another, this is commonly accomplished through method invocation. Method invocation is an expression in which one object supplies another with information (in the form of arguments), and the second supplies the first with other information (in the form of the return value). These mechanisms are the major means of inter−object communication and coordination. Of course, method invocation can also be used within an object, allowing one part of the object to communicate with another.

We have previously seen how interfaces specify methods that an object provides. Now, we turn to the question of how method behavior is actually implemented. Statements provide the key. Performing a method amounts to following the instructions associated with that method, i.e., stepping through the instructions for that rule. Statements are the steps of those instructions. By sequencing statements, you can build a rule that the computer can follow to accomplish a desired task. Some rules require information in order to accomplish their tasks. (For example, a rule that doubles a number needs the number to be doubled.) Some rules produce results. (For example, the doubling rule might produce the doubled number.) Some rules behave differently under different circumstances. (This uses a conditional statement).

In order to use a rule — to interact with it — you need to know whose rule it is, what information you need to supply in order for the rule to do its work, and what the rule will give you in return. This prefigures the idea of method signature. There are other things you'd like to know about a rule — such as the relationship between the rule's input and its output — and these form the basis of the rule's documentation.

For example, here is a rule for printing a brief form letter:

```
to printFormLetter using (String title, String firstName,
                          String lastName)

  1. print "Dear "
  2. if (title isn't null) print title + lastName
     else print firstName
  3. println ":\nWe are tremendously pleased to inform you that"
  4. println "you have won!".toUpperCase()
  5. println "Not much, but what did you expect?"
  6. println " Sincerely,\n me"
```

It's just a short hop from this pseudocode rule to real Java:

```
void printFormLetter(String title, String firstName,
                     String lastName) {
    if (title != null) {
        Console.print(title + lastName);
    } else {
        Console.print(firstName);
    }
    Console.print(":\nWe are tremendously pleased "
                    + "to inform you that ");
    Console.println("you have won!".toUpperCase());
    Console.println("Not much, but what did you expect?");
    Console.println("                Sincerely,\n"
                  + "                    me");
}
```

6.6.1 Method Invocation Execution Sequence

Method invocation is, as we have seen, an expression. To invoke the *printFormLetter*, we need to know whose method it is. We follow this object expression with a dot, then the name of the method, then the parentheses−enclosed parameter list:

```
theWidgetCompany.printFormLetter("Prof.", "Pat", "Smith")
```

To evaluate this expression, we need to invoke *theWidgetCompany*'s *printFormLetter* method (using the rule, or instructions, or method body, provided above) with the arguments “Prof.”, “Pat”, and “Smith”.

The first step in method invocation is parameter binding. In this step, each parameter name (*title*, *firstName*, and *lastName*) is treated as though it were newly declared and it is given the value of the corresponding argument. (Recall that parameters are the *names* in the method declaration, while arguments are the *values* supplied in the method invocation expression.) In order for this to work, each value must be assignable to the corresponding parameter's declared type.

After parameter binding, method invocation proceeds as though the method body were a simple block. The block is, however, within the scope of the parameter bindings, so that inside the block the parameter names can be used to refer to the provided argument values. For example, in the body of the *printFormLetter*, *title* is bound to “Prof”, *firstName* is bound to “Pat”, and *lastName* is bound to “Smith”.

Now the body statements are executed in turn. In this case, the first statement is an if, so its test expression is evaluated to determine whether to execute the consequent block or the alternative block. When the test expression

```
title != null
```

is evaluated, title is bound to “Prof”, so it is not `null`, causing the consequent to execute.

This argument−value−providing is one way in which method invocation implements inter−entity communication: the value is communicated from the method−invoker to the method owner.

6.6.2 Return

This special statement can only be used inside method bodies. It is used to terminate the execution of the method body. It is also what is responsible for making a method body — which is essentially a block statement — return a value — which is a necessary property of a method invocation expression (unless the method's return type is `void`).

The need for this statement arises when the sequence of instructions that you are writing is turned into a method body. In this case, you need to say what the method *returns*. This return value becomes the value produced by evaluating a method invocation expression. This is accomplished using a ***return statement***. The syntax of a `return` statement is

Syntax of a `return` statement
`return` *`expression;`*

where *expression* can be any arbitrary Java expression. Remember: the `return` statement — a statement — does not have a value, but the method invocation − an expression — does.

To execute a `return` statement, evaluate the expression. Then, exit the enclosing method, providing the value of the expression as the return value of the method invocation expression. Exiting the enclosing method means both exiting from the block that is the method body and also exiting the scope of the parameter/argument bindings.

After a `return` statement, execution proceeds at the method invocation whose method body contained the `return` statement; evaluation of this expression is complete (with its value the value supplied by the `return` statement) and execution of the statement containing the method invocation continues.

For example, if we execute

```
String transformed = this.transform("Knock, knock");
```

and the transform method of this object ends with the line

```
return "Who's there?";
```

then the value of the invocation **`this.transform("Knock, knock")`** is `"Who's there?"`. Execution continues by assigning the value of the invocation (`"Who's there?"`) to the name *transformed*.

Another example is the *doDouble(int)* method mentioned above. The code for *doDouble* might read:

```
int doDouble(int whatToDouble) {
    return whatToDouble * 2;
}
```

To evaluate the application of *doDouble* to 7:

1. The parameter name *whatToDouble* is bound to 7.

2. Within the scope of this binding, the body block of *doDouble* is executed.

    a. Each statement in the block is executed in turn. Since there is only one statement, it is executed.

        i. The expression whose value is to be returned is evaluated. This requires evaluating the subexpressions (name *whatToDouble* and literal `2`) and then applying the operator to these values.

        ii. The value produced by the operator expression (`14`) is returned by the method.

3. This exits both the method body block and the parameter scope, providing the value (`14`) as the value of the method invocation expression.

There is also an alternate form of `return` that does not take an expression. This form is

used in methods whose return type is ***void***. In this case, a `return` statement executes by exiting the method (and, with it, the scope of the parameter names). Since the simple `return` statement is used only in methods whose return type is `void`, there is no value for it to supply.

This `return` statement can also be left implicit certain methods. For example, in the *printFormLetter* method that we saw above, there was no explicit `return` statement. In Java, a method without a `return` statement is presumed to have a `return` statement as its final statement. This `return` statement is a simple **`return;`** — it is the form that does not return a value. So the end of that method body was equivalent to saying

```
    // ...
    Console.println("                 Sincerely,\n"
                    + "                  me");
    return;
}
```

In a method whose return type is not `void`, an explicit return statement must always be executed in order to provide the method's return value. Value−returning is another example of inter−object communication.

Chapter Summary

- Statements combine expressions to produce useful behavior.
- A statement does not have a value or a type.
- A statement is executed to produce an effect.
- A side−effecting expression followed by a semicolon is a simple statement.
- Declarations and definitions are also simple statements.
- A sequence of statements can be grouped into a block by surrounding the sequence with braces { }.
- Conditional statements allow you to write code containing alternative execution sequences. The execution sequence of a conditional statement depends on the result of evaluating a boolean expression.
- A loop allows the same block of code to be executed repeatedly, until an exit condition — a boolean expression — is true.
- A `return` statement is used to exit from a method, with or without a value.
- Method bodies, or rules, use sequenced statements — including loops and conditionals — to produce chunks of executable behavior. A method is specified by its name, the information it needs, and the value (if any) that it produces.

Exercises

1. Using Java's `if` statement, write instructions for determining which team returns an out-of-bounds ball to play in a soccer game. In soccer, the team that did not last touch the ball receives possession of the ball and returns it to play.

   a. You may presume that you have a method, *lastTouch*, that returns either *homeTeam* or *visitTeam*, and that the goal of your code is to assign the correct team value (either *homeTeam* or *visitTeam*) to the already-defined name *possessingTeam*.

   b. In addition, make your code determine whether *returnBallToPlayMethod* is *sideThrow*, *cornerKick*, or *goalKick*. You may make use of the *ballOutLine* method to determine whether the ball exited via the *sideLine*, the *homeEndLine*, or the *visitEndLine*.

      [Footnote: If the ball has exited via the side line, the return is by side throw. If the ball exits via the home end line and is last touched by the home team, the visitors return the ball to play by means of a corner kick. A ball that is pushed beyond the home end line by the visiting team is returned by the home team via a goal kick. The situation at the visitor's end line is the opposite.]

2. Using Java's `while` statement, give instructions for building a tall tower of blocks.

3. Using Java's `while` statement, give instructions for blowing up a balloon.

4. Which of the following are expressions, which statements, and which illegal? For the expressions, indicate the type and value. For the statements, indicate the effect (if known) and the execution sequence. You may assume that *x* is an `int`, *b* a `boolean`.

   a. `int x = 5`
   b. `boolean b;`
   c. `x + 3`
   d. `x = x + 3`
   e. `x = x + 3;`
   f. `x == 3`
   g. `x == 3;`
   h. `b = x == 3;`
   i.
   ```
   {
      Console.print("What is your name? ");
      String name = Console.readln();
      String cap = name.toUpperCase();
   }
   ```

5. What will the value of *d* be after each of the following statements? Also, indicate any other changes that may occur as a result of executing the statement. You may assume that they are executed in the order given.

   a. `double d = 3.5;`

   b. `d = d * 3;`

   c.
   ```
   if (d < 8) {
        Console.println("d is pretty small");
   }
   ```

   d. `d = 2.0;`

   e.
   ```
   while (d < 30) {
        d = d * 2;
   }
   ```

Interlude B

Expressions and Statements

Chapter Overview

This interlude explores what you've learned so far about Java expressions and statements. There is a supporting executable, distributed as a part of the on−line supplement to this book, which allows you to experiment with the ideas described here. The goals of this interlude are to show you how expressions and statements can be used in context and simultaneously to give you an opportunity to explore interactions among entities and how these interactions can be used to generate a variety of basic behaviors.

Objectives of this Chapter

1. To increase familiarity with expressions and statements, including `return` statements and conditionals.
2. To be able to read and write simple sequences of instructions.
3. To appreciate how multiple independent instruction−followers can produce behavior through their interactions.
4. To begin to appreciate the differences between parameters, local variables, and fields.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

B.1 The Problem

This interlude is inspired by a simple child's toy called an Etch A Sketch®. In case you may not be familiar with an Etch A Sketch®, here is a brief description:

[Footnote: The Etch A Sketch® product name and the configuration of the Etch A Sketch® product are registered trademarks owned by the The Ohio Art Company. Used by permission.]

An Etch A Sketch® is a rectangular frame (generally red) with a silver screen in the center and two white knobs, one in each of the lower corners. Inside the silver screen is a point of darker grey:

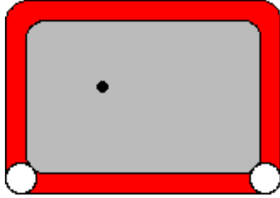


Figure 1: A simple drawing tool.

An Etch A Sketch® is a simple drawing tool. Turning the knobs moves the darker grey point around the screen. As the point moves, it leaves a darker grey trail behind it, showing where it has been. By coordinating the motion of the knobs, you can draw pictures.

On an Etch A Sketch®, each knob controls one direction of motion of the darker grey dot. Rotating the left knob moves the dot from side to side. Rotating the right knob moves the dot up and down. Keeping a knob still prevents the dot from moving in the corresponding direction. So the position of the knob determines the position of the dot on the screen (in one dimension) and changing the knob position moves the dot (in that dimension).

By rotating just one knob — by leaving the position of the other knob fixed, or constant — you can draw a straight (horizontal or vertical) line, as in figure 2. By rotating both knobs at appropriately coupled velocities, you can draw diagonal lines of varying slope. Proficient Etch A Sketch® users can draw complex pictures by coordinating the knob position changes.

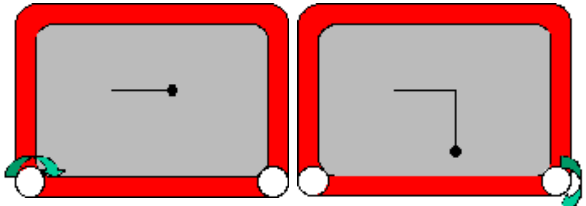


Figure 2: Drawing a straight line by rotating the appropriate knob.

In this interlude, we will explore the instructions required to perform similar operations on a similar (though less brightly colored) display. In our application, we will supply a rule that describes the current position of each knob. For example, the position might be constant — always the same — in which case, the dot wouldn't move. Or it might be steadily increasing, in which case the dot would move steadily across the screen.

Each rule will be read (and executed) repeatedly. It is as if, behind the scenes, an instruction−follower were to check the position of each knob continually and update the position of the dot correspondingly. Each time that the instruction−follower wants to know what to do with the dot, it will ask our rule. If the rule always gives the same value, it will be as though that knob is stuck in one position. If the rule changes the value, the same change will be made to the knob's position over and over again. So, for example, if the rule says “increase the current position,” the dot will move across the screen until it reaches the edge.

In fact, there will be two instruction−followers, one for each knob. In addition, they're not guaranteed to run at the same speed, or even to take fair turns. So, even if both knobs were to have the same rules (e.g., “increase the current position by 1”), we might discover that our dot was moving twice as fast horizontally as vertically.

Question: What would the resulting picture look like?

Answer: It would slope upwards gradually. @@supply pic.

Or the horizontal knob instruction−follower might check its rule three times, then the vertical knob once, then the horizontal knob once, then the vertical knob three times, then both at once.

Since the knobs are being checked by independent instruction−followers, any schedule is possible in principle. By observing the actual behavior of the system, we can try to write rules to coordinate the behavior of the two knobs explicitly. To begin with, we'll just assume that they run at about the same rate.

An interesting feature of our program is that the knob−rules don't have any way to tell which knob they're controlling. The rules just say things like “turn the knob to a higher value” or “turn the knob lower” or “set the knob to the middle” (or “...halfway to the

edge”).

But on to the details....

B.2 Representation

In our application, instead of actually rotating knobs, we will represent a knob position as a number. We will use a standard Cartesian coordinate frame, with (0,0) in the center, increasing horizontal coordinates to the right, and increasing vertical coordinates at the top.

[Footnote: In a later chapter, we'll see computer graphics that use a different coordinate system, sometimes known as “screen coordinates.”]

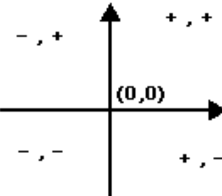


Figure 3: Cartesian coordinates.

The value of the knob position can range between `-maxPos` and `maxPos`. Since (0,0) is dead center on the screen, (0, `maxPos`) is the center top of the screen, while (`-maxPos/2`, 0) is middle height, halfway to the left edge. The actual size of the Etch A Sketch® window (and therefore the value of `maxPos`) will vary as the window is resized.

[Footnote: In particular, `maxPos` may have different values in the horizontal and vertical rules. `maxPos` is simply the largest coordinate in whichever dimension the rule controls.]

Question: What four points represent the four corners of the Etch A Sketch® window?

Answer: (–maxPos, –maxPos), (–maxPos, maxPos), (maxPos, –maxPos), (maxPos, maxPos). These are lower left, upper left, lower right, and upper right in order. But note that maxPos in the vertical dimension may not have the same value as maxPos in the horizontal dimension, and also that the value of maxPos may change as the window is resized.

Our job will be to write the instructions for the next knob position: a rule that returns the desired dot position. We'll need one rule for the horizontal knob and one for the vertical, of course.

The form of a control rule is a sequence of Java statements ending in a statement of the

form

```
return double;
```

where *double* is some Java expression with type `double`. The value returned by your control rule will be used as the new position of the dot. So, for example,

```
return 0;
```

is a rule that holds the knob in the center, i.e., keeps the dot in the middle of the screen. (Why is it ok to return 0 rather than 0.0?)

Remember, though, that this is just one rule. The other rule might be moving the dot along steadily, or holding it at the edge of the screen, or doing something else entirely. If `return 0;` is the vertical rule, then the dot will remain halfway up the screen, but it could be anywhere along that halfway line.

Question: What rule would keep the dot centered horizontally (halfway between the left and right edges) on the screen? Is this a horizontal− or a vertical−knob control rule?

Answer: `return 0;` *as a horizontal−knob control rule. Using this rule, the dot may move up and down (depending on the vertical control rule), but will always be halfway between the two sides of the screen. Its horizontal position is fixed, i.e., constant.*

horizontal rule
return 0;

If you use this rule as a vertical control rule, it will keep the dot halfway up the screen. Its side−to−side (horizontal) position would be determined by the horizontal control rule.

vertical rule
return 0;

Question: How would you position the dot almost in the upper right−hand corner? The answer should involve a horizontal rule *and* a vertical rule.

Answer: The horizontal rule should position it at the far right of the screen. Examining the coordinate system (figure ??), we see that this is at maxPos. So `return maxPos;` *as a horizontal−knob control rule. The vertical control rule should position the dot at the top of the screen, which is maxPos in that dimension. So the vertical rule is also* `return maxPos;`

horizontal rule	vertical rule
return maxPos;	*return maxPos;*

Question: How about the upper left−hand corner? Again, the answer should involve a horizontal rule and a vertical rule.

Answer: In this case, the horizontal rule should put the dot at the far left, which (according to the coordinate frame in figure 3) is at −maxPos. So the horizontal rule should be `return -maxPos;` *The vertical rule should put the dot at the top, which is still maxPos, so the vertical rule should be* `return maxPos;` *as in the previous question.*

horizontal rule	vertical rule
`return -maxPos;`	`return maxPos;`

The vertical rule has not changed. But because it is interacting with a different horizontal rule, the on−screen behavior is different.

Question: Assume that you have rules that position the dot in the upper left−hand corner. Now suppose that you swap the horizontal and vertical rules. Where is the dot now? What if you use the horizontal rule for both horizontal and vertical behavior?

Answer: This question involves swapping the horizontal and vertical rules from the previous question:

horizontal rule	vertical rule
`return maxPos;`	`return -maxPos;`

Note that it is the vertical rule that now returns a negative value. Examining the coordinate frame in figure 3, we see that the horizontal rule puts the dot on the right−hand edge of the frame, while the vertical rule puts the dot at the bottom. So the result will be a dot in the lower right−hand corner of the frame. Swapping the rules — so that the same two rules play different roles — changes the behavior of the system, too.

A rule is a sequence of statements including a value returning statement. A rule is executed by following the instructions until a return statement is reached. This kind of rule is essentially the body of a Java method. This application focuses attention on the sequence of statements — the method body — but its ideas apply to Java methods as well.

B.3 Interacting with the Rules

In introducing this system, we said, “Each rule will be read (and executed) repeatedly.” But nothing about the observable behavior of the system has really made this clear. In fact, in the examples above, each rule might be followed only once and the same results would still be produced. Each of the rules that we have used so far has been like setting the Etch A Sketch® knob to a certain position. If you set the knob to the center, it looks

the same whether you do it once (and then just leave it there) or do it over and over, repeatedly making sure that the knob is set to the center position. So how can we tell what the Etch A Sketch® is actually doing?

In order to see that the rules are executed repeatedly, we need another (non−rule) way to change the position of the dot. If we could move the dot this way, we would see that continued execution of the rules forces it back to the "stuck knob" position described by the rules. Imagine that both rules say

```
return 0;
```

So each instruction−follower will keep putting the dot right in the middle of the screen. This won't look like much until we move the dot. But if we get the dot to jump to the top of the screen, the next time that the instruction−follower executes the rule it will still put the knob back into the position mentioned in the rule — the center of the screen, in our example. We can see that the rules for our Etch A Sketch® are invoked repeatedly, rather than just run once, by interacting with them.

In fact, in our Etch A Sketch® application, you can move the dot using your mouse. By clicking the mouse at a particular point on the Etch A Sketch® screen, you move the dot there. Almost immediately, the instruction−followers go back to their knob−rule−checking. This means that if you position the dot with the mouse, it may not stay there for very long. Fortunately, you can see where the dot has moved because every time that the instruction−follower moves the dot, it leaves a (red) trail behind it. Indeed, we will use this feature to draw pictures. So, for example, you can make a line from the upper lefthand portion of the Etch A Sketch® to the center by using the rules

horizontal rule	vertical rule
`return 0;`	`return 0;`

and clicking the mouse in the upper lefthand portion of the window:

@@add pic

The mouse click moves the dot momentarily, then the execution of the rules brings the dot back to the center, leaving a trail. You — the user — provide a third independent control to this application. By interacting with the dot, you, too, affect its behavior.

Question: Combining these two observations — leaving trails and "jumping" the dot around using the mouse — can you figure out how to create an asterisk (a bunch of line segments intersecting in the center)?

Answer: If you run the Etch A Sketch® with both rules set at `return 0;` *the dot will return to the middle, no matter where you move it with the mouse:*

horizontal rule	vertical rule
`return 0;`	`return 0;`

Now, with the mouse, click anywhere. The dot will jump there, then immediately (well, as soon as the horizontal and vertical rules are sampled) return to the center. This should draw a line from wherever you clicked to the center. Repeat this, clicking somewhere else. Eventually, you'll have a lovely asterisk.

@@add a picture.

Actually, you can make an asterisk around any point in the screen, not just the center of the screen. Try setting the rules to:

horizontal rule	vertical rule
`return maxPos/2;`	`return 0;`

Then repeat the clicking around process. Now you should get an asterisk centered at (maxPos/2, 0), i.e., centered vertically but in the right half of the screen.

@@another pic.

Rule bodies as we have presented them here are really just very simple method bodies. The return statement supplies the return value of its containing method. In the rule form that we are using here, the enclosing method declaration and braces are omitted. We will see more of how to write and use methods in the next chapter.

By manipulating the behavior of each method, the role played by each, and how we and the system interact, we can generate a variety of different behaviors even with extremely simple code.

B.4 Paying Attention to the World

So far, the rules that we have written return the same value, no matter where the dot starts out. This corresponds to a rule that drives the knob to a certain fixed position, regardless of where it starts out. When you don't move the dot with the mouse, the rule causes the dot to sit still. When you do use the mouse to move the dot, the rule causes the dot to jump back to the same place that it has been. In this section, we will see how our rules can respond to information that they are given.

Each time that an instruction−follower goes to execute a rule, the name *pos has been pre−defined for it to hold the current position of the dot (along the relevant dimension). So, if the dot is half−way between the left side of the screen and the center, pos will be* `-1/2 maxPos` *when the horizontal rule is invoked, while if the dot is all the way at the right side of the screen, pos will be* `maxPos`*.*

Question: Using this information, write a rule that causes the dot to stay where it is.

Answer: "Where it is" is always pos — pos is the current position of the dot when the rule is about to be executed. So, no matter where the dot is, we can make it stay there using the rule

```
return pos;
```

If the dot is at 36, pos will be 36 and this rule will return 36. If we click the mouse and move the dot to 78, pos will be 78 the next time that the rule is executed, and we will return pos, or 78. Since pos is always where the dot is, returning pos will keep the dot there.

Note that *pos* is defined anew each time that the rule is executed. It is, in effect, a parameter to the rule.

Question: What happens when the dot is moved, using the mouse?

Answer: When the dot is moved, the rule still says "stay where you are." So each time you click the mouse, the rule adjusts the knob to keep the dot where you've put it.

Note that the horizontal rule and the vertical rule each have their own version of *pos* . So *pos* in the horizontal rule has nothing to do with *pos* in the vertical rule; each gets its own proper position.

Now, using the information that *pos* is where the dot is, we can cause the dot to move and keep moving. Each time the knob is checked — each time the rule is invoked — the knob should turn just a little bit.

Question: What would such a rule look like?

Answer: We could use a rule that says

```
return pos+1;
```

This rule checks where the dot is, then instead of setting the knob there, it moves the knob slightly. The next time the rule is executed, it will move the dot a little bit further over. This will continue to happen until the dot reaches the edge of the screen.

[Footnote: Why not **`pos = pos + 1`**? *Pos* is a local name that this piece of code has for the current position of the dot. It is NOT the "control" for the current position of the dot. When the instruction−follower is about to execute the instructions, it creates a new dial — called *pos* — and sets it to the value that represents the current position of the dot. After this happens, there's no additional connection between the value of the *pos* dial and the position of the dot. Reading the *pos* dial tells you what the current position of the dot

is. But changing the *pos* dial doesn't move the dot. Returning a value does.

What would happen, then, if we ran with the horizontal rule **`pos = pos + 1`**? First, we'd get an error. Remember, a rule has to end with **`return double;`** So now consider **`pos = pos + 1; return 0;`** This would be exactly the same as just return 0;, i.e., it would drive the dot to the center. How about **`pos = pos + 1; return pos;`**? This works, but it isn't as “nice” as **`return pos+1;`** The reason it works is that it first modifies the *pos* dial to have the value we want to return (**`pos+1`**), then returns the value on the dial. There's really no reason to modify the dial; we can just return the value directly.]

The name *pos* is a ***parameter*** — a name whose value is defined before the rule (method) body begins to execute. We can create names (dials) of our own as well. For example, a much more long−winded way of writing the previous rule might be:

```
double velocity = 1;  // how much the position changes by.
double newPos;        // what the new position will be.

newPos = pos + velocity;  // compute next position ...
return newPos;            // ... and return it.
```

The names velocity and *newPos* here are new local variables we create. Their declarations last until the return statement. Each time that this set of instructions is executed, the declaration line `double velocity = 1;` is executed again, and a new dial called velocity is created. (Yes, that's a lot of wasted dials. Don't worry; Java has facilities to make sure they are recycled.) In a later section, we will see a different kind of name that persists from one rule execution to the next.

Question: What happens when the dot reaches the edge of the screen?

Answer: At this point, pos will continue to be increased. But values greater than `maxPos` *aren't allowed. (The application is written so that values greater than maxPos are treated just like* `maxPos`*, so the dot will sit at the edge of the screen.)*

Question: How would you make the dot move in the other direction?

Answer: With a rule that says

```
return pos-1;
```

This rule makes the knob turn a little bit in the opposite direction. Remember, returning a value is the way to move the dot.

B.5 Fancy Dot Tricks

Question: What would happen if you used the rules:

horizontal rule	vertical rule
`return pos+1;`	`return 0;`

Assume that the dot starts in the center of the screen.

Answer: You would get a horizontal line from the center of the screen to the right hand edge of the screen.

Question: How would this be different if the rules were

horizontal rule	vertical rule
`return pos+1;`	`return pos;`

Answer: If the dot starts in the center of the screen and you don't click the mouse, you wouldn't be able to tell whether the vertical rule said return 0; or return pos; The value of pos (for the vertical rule) would start out as 0, and since nothing changes it, it would remain 0. The only way to see a difference is to move the dot (using the mouse). If you let the dot move across the screen until it's halfway to the right edge, then click the mouse in the lower left (at the X), here's what you'll see:

@@add picture

Question: What does the dot do if you start in the lower left hand corner of the screen and use the rules

horizontal rule	vertical rule
`return pos+1;`	`return pos+1;`

Answer: This rule pair would draw a diagonal line from the lower left hand corner of the screen towards the upper right hand corner.

[Footnote: Actually, the line would only go towards the corner if the screen were relatively square. This diagonal line has a slope of 1.]

Question: Can you make the dot move from the `maxPos` edge of the screen to the other edge when it gets there?

Answer: In order to do this, we need to check whether we've gotten to the `maxPos` *edge. We can do this using an if statement:*

```
if (pos < maxPos) {
    return pos + 1;
} else {
    return -maxPos;
}
```

Question: Can you make the dot move more quickly across the window?

Answer: In the previous rules, we've increased pos by 1 each time. If we increase pos by a larger number, it will move more quickly. In fact, this increase to pos is the velocity — the speed — of the dot.

We can use this rule to create a sort of barber−shop pole effect — a slowly climbing spiral around the window. To do this, we use a horizontal wrap−around rule and a vertical wrap−around rule. By setting the horizontal rule to move more quickly than the vertical rule, we get a line with a gradual slope. Since we're using wrap−around rules, the line repeats over and over again as it moves up the screen.

So starting in the lower left hand corner of the screen and executing

<table>
<tr><th>horizontal rule</th><th>vertical rule</th></tr>
<tr><td><pre>if (pos < maxPos) {
    return pos + 5;
} else {
    return -maxPos;
}</pre></td><td><pre>if (pos < maxPos) {
    return pos + 1;
} else {
    return -maxPos;
}</pre></td></tr>
</table>

produces something like:

@@add pic. a sequence would be better.

For each Etch A Sketch® rule, *pos* is a name whose value is fresh each time the rule is executed. There is no connection between the value of *pos* from one invocation of the horizontal rule to the next. The value of *pos* for the horizontal rule is unrelated to the value of *pos* for the vertical rule. This behavior is essentially the behavior of a parameter to a Java method. In the next section, we will see a different kind of name.

B.6 Remembering State

In the previous section, we saw how to prevent the dot from getting stuck at one edge of the screen by jumping it to the other edge. It might have been nice to have the dot bounce back from the edge — turning to move in the opposite direction — instead. It turns out that that behavior requires an additional idea and a corresponding bit of machinery.

Suppose that we wanted to get the dot to turn around. We might start with a rule that looks like the “jump to the other edge” rule, trying to detect when we've bumped into the *maxPos* edge:

```
if (pos < maxPos) {
    return pos + 1;
}
```

This rule seems reasonable enough. It will cause the dot to move along until it reaches maxPos. But what then? When we reach maxPos, the if test will fail and we'll drop through to the else clause. It goes through `maxPos - 2`, `maxPos - 1`, `maxPos`. Now, it needs to go to `maxPos - 1`, (and then to `maxPos - 2`, `maxPos - 3` and so on). So we might try

```
else {
    return pos - 1;
}
```

Sure enough, the dot's positions will be `maxPos - 2`, `maxPos - 1`, `maxPos`, `maxPos - 1`. But then what? The problem is that when the dot is at `maxPos - 1` and this rule is executed again, the if test will succeed! The next position of the dot will be (`(maxPos - 1) + 1`), or `maxPos`! Then the if test will fail, triggering the else clause: `maxPos - 1`. At this point, the dot will oscillate between `maxPos - 1` and `maxPos` forever.

What went wrong? As always, our errors are informative. The problem is that the condition we're testing — whether our position is < maxPos — doesn't really tell us what we need to know — which direction to move in. Our position might be `maxPos - 1` because we're heading towards `maxPos`, or it might be `maxPos - 1` because we're heading back towards `maxPos - 2`. The if test doesn't give us any way to tell the difference. In fact, nothing about the current rule execution or our current position can answer this question for us. Instead, we need to know something about the *previous* execution, or about where we've been.

B.6.1 Fields

At this point, we need to introduce some new machinery. In our application, there is a special box (for each dimension) where we can enter names that persist from one execution of a rule to the next. These names correspond to fields of instance objects. They are like airport lockers: places that you can leave things when you're executing the rule and find them the next time you come back into town.

There are several different ways we can use airport lockers to solve this problem. The simplest is probably just to remember which direction we're going in. We can do this using a boolean name. In this case, we'll call the boolean increasing. We start with increasing true. So the declaration should say:

fields
boolean increasing = true;

(Recall that a declaration follows the *Type−of−thing Name−of−thing* rule. So this declaration says we have a boolean — a true−or−false kind of dial — that is called increasing. Because this is a definition, not just a declaration, it also sets the dial to read true.)

Now, we can write an `if` statement that says what to do if we're increasing: increase pos, unless we've hit the edge.

```
if (increasing) {
    if (pos < maxPos) {
        // Keep going higher — return the next position
        return pos + 1;
    } else { // Not (pos < maxPos)
        // We've hit the edge — turn around
        increasing = false;
        return pos;
    }
}
```

The `else` condition is similar, but with the signs reversed:

```
else { // Not (increasing)
    if (pos > - maxPos) {
        // Keep going lower — return the next position
        return pos - 1;
    } else { // Not (pos < maxPos)
        // We've hit the edge — turn around
        increasing = true;
        return pos;
    }
}
```

Question: Can you write a similar rule that relies on a numeric piece of state — double previousPos — instead? What does the declaration of persistent state look like? What happens the first time the rule is executed?

Answer: First, declare a field (in the special box):

fields
double previousPos = 0;

Note that the code starts by checking the boundary cases. If we're at the edge, we need to go inwards. Otherwise, we remember where we were this time (previousPos=pos;) and return a number that continues moving the dot in the appropriate direction.

```
if (pos >= maxPos) {
    //we're at the higher edge.
    previousPos = maxPos;
    return maxPos - 1;
} else if (pos <= -maxPos) {
    //we're at the lower edge.
    previousPos = -maxPos;
    return maxPos + 1;
} else if (pos > previousPos) {
    //we're moving up: return a higher number.
    previousPos = pos;
    return pos + 1;
} else { // (pos <= previousPos)
    //we're moving down; return a lower number.
    previousPos = pos;
    return pos - 1;
}
```

B.6.2 Fields versus Local Variables

Consider the previous example: using previousPos to keep track of which direction we were going. Why do we need previousPos here? Why can't we just use pos? There are two reasons. First, *pos* is already being used for something — the current position of the dot. But the other reason is that *pos* gets a new value each time this set of instructions is executed. (Actually, *pos* gets re−created each time this set of instructions is executed.) So, if we put something we want to remember into pos, it won't be there the next time that these instructions are executed. We need to create a special value — a field — to hold things that we want to remember from one execution of these instructions to the next.

Field declarations must be made in the special box. Declarations in the regular code box are allowed, but they do not carry over from one execution to the next. Instead, a name declared in running code is a temporary scratch space. The corresponding dial or label is created each time that the declaration is executed (as a part of following those instructions) and discarded when the return statement is reached. Such local scratch space is called a ***local variable***.

Contrast this with the use of *velocity* and *newPos* in an earlier section:

```
double velocity = 1;  // how much the position changes by.
double newPos;        // what the new position will be.

newPos = pos + velocity;  // compute next position...
return newPos;            // ... and return it.
```

The names *velocity* and *newPos* here are local variables. They are *not* fields. That is, they are new dials that are created each time the rule is executed — local scratch space that only exists during a single rule execution — and they go away when the rule execution is done. Next time the rule is executed, they will be recreated. In contrast, a field — like previousPos in the rule above — sticks around from one rule execution to another.

Chapter Summary

In this chapter, we have seen simple pieces of Java code that produce behavior. Each short set of instructions is in effect the body of a Java method; a value is returned at the end. The behavior of the system as a whole depends on the particular methods written. In addition, system behavior depends on how those methods are coupled together and how you as a user interact with them.

There are three different kinds of names that can be used in your code. First, you can use names that have been pre−defined for you, like *pos* . These are called ***parameters***. In other chapters, we will see that in a Java method, all parameter names are included in the method declaration.

There are also two kinds of names that are declared using standard declarations or definitions. One kind is a temporary name that can be used during a single application of your rule. These names can be declared anywhere in your code. They are called ***local variables***. The names *newPos* and *velocity* are examples of local variables. Local variables can be declared inside a method.

The last kind of name sticks around from one use of your rule to another. These names must be declared in a special box, separate from your rule code, but can be used freely in your rule code. These names are called ***fields***. In this chapter, *increasing* is an example of a field. In the Etch A Sketch®, fields are declared in a separate box. In Java code generally, fields are declared outside of methods (but within an enclosing class).

Exercises

See the text for things marked with a **Question:**. Also:

1. Implement constant acceleration. Velocity is the change in position over time. For example, the rule `return pos + 1;` has a velocity of 1, while the rule `return pos - 5;` has a velocity of 5 in the opposite direction. Acceleration is the change in velocity over time. For example, if we `return pos+1;` when the rule is executed the first time and then we `return pos+3;` when the rule is executed the second time, the change in velocity (i.e., the acceleration) is 2. To implement a constant acceleration, you need to change the velocity by the same amount each time. This means that the rule can't return *pos* + a constant; instead, it has to return *pos* + an amount that changes each time. (Hint: Use a field.)

2. Can you make the dot go in a parabolic path? (Hint: what accelerations does it need?)

3. We have given you a parameter named *otherPos*. Each time that a rule is followed, *otherPos* begins with the position of the dot along the *other* axis. Using this information, implement a function plotter. Write the code to plot the following:

   - *y = x^2;*
   - *y = sin(x);*
   - *y = 1/x;*

   You may want to look at the `Math` library.

Chapter 7

Building New Things: Classes and Objects

Chapter Overview

- How do I group together related rules?
- How do I build a computational object?
- What are Java programs *really* made of?

In this chapter, you will learn to put together the pieces you've already seen — things, names, expressions, statements, rules, and interfaces — to create computational objects that can populate your communities.

In order to create an individual object, you first have to describe what kind of object it is. This includes specifying what things you can do with it — as in its interface(s) — but also how it will actually work. This description of the “kind of object” is like building a recipe for the object, but not like the object itself. (You can't eat the recipe for chocolate chip cookies.) These object−recipes are called ***classes***.

For each thing that your object can do, your class needs to give a rule−recipe. This is called a ***method***. Your objects may also have (named) pieces. These are called ***fields***, and they are special Java names that are always a part of any object made from this recipe.

When you actually use your class (recipe) to create a new object, there may be things that you need to do to get it started off right. These startup instructions are called a ***constructor***.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

When you are building an object, you are bound by the interfaces it promises to meet. If the interface promises a behavior, you have to provide a rule (method) body for the object to use.

This chapter is supplemented by reference charts on the syntax and semantics of Java classes, methods, and fields. It includes style sidebars on good documentation practice.

Most of the syntax of this chapter is covered in the appendix Java Charts.

Objectives of this Chapter

1. To recognize the difference between classes and their instances.
2. To be able to read a class definition and project the behavior of its instances.
3. To be able to define a class, including its fields, methods, and constructors.

7.1 Classes are Object Factories

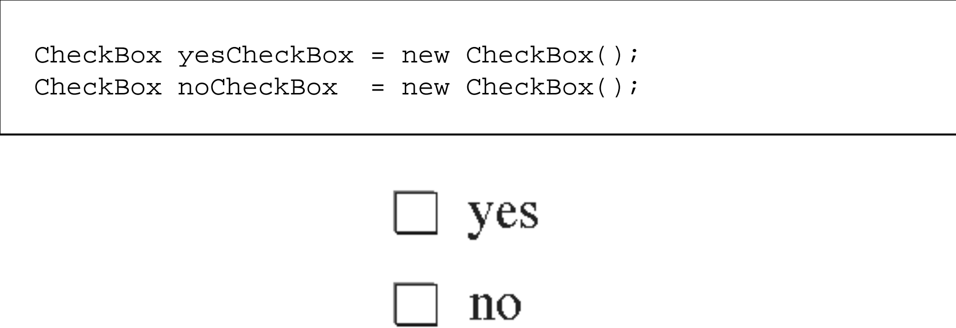
In Chapter 4, *Specifying Behavior: Interfaces*, we saw how to build an interface, or specification, that described the contract a particular kind of object would fulfill. We also saw that an interface does not provide enough information to actually create an object of the appropriate kind. Interfaces do not say anything about how methods actually work. They do not talk about the information that an object needs to keep track of. And they do not say anything about the special things that need to happen when a new object is created.

In this chapter, we will learn how to create objects and how to describe the ways in which they work. The mechanism that Java provides for doing this is called a ***class***. Like an interface, a class says something about what kind of thing an object is. Like an interface, a class defines a Java type. However, **interfaces specify only contracts; classes also specify implementation**. Class ***methods*** are full−fledged rules, with bodies telling how to accomplish the task of that rule (not just the rule specification, or method signature, of abstract interface methods). Classes also talk about data — information to be kept track of by objects — as well as methods, or behavior. And a special part of a class — the ***constructor*** — talks about how to go about creating an object of the type specified by that class.

7.1.1 Classes and Instances

Objects created from a class are called ***instances*** of that class. For example, the class CheckBox refers to the instructions for creating and manipulating a GUI widget that

displays a selectable checkbox on your computer screen. *CheckBox* is the name of the class, i.e., of the instructions. Let's say we create two particular checkboxes:



```
CheckBox yesCheckBox = new CheckBox();
CheckBox noCheckBox  = new CheckBox();
```


Figure 7.1. The actual CheckBoxes.

The two objects labeled by the names *yesCheckBox* and *noCheckBox* are instances of the class CheckBox. That is, they are *particular* CheckBoxes. The instructions for how to create — or be — a CheckBox, on the other hand, aren't a CheckBox at all; the instructions are *instructions*, or a class. In fact, the instructions are an object, too, though a very different kind of object and not one as obviously useful as a CheckBox or a Timer or a Counter. The kind of object the instructions are is called a ***Class***.

Because the class contains the instructions for how to make a new instance and for how to behave like an instance of that class, we sometimes say that a class is like a **factory** where instances are made. Both a factory and its product are objects, but factories and the widgets that they make are very different kinds of objects. The factory has all of the know−how about its instances. But the factory isn't one of its instances, just as the **class** CheckBox isn't a CheckBox. It's a **factory**!

7.1.2 Recipes Don't Taste Good

Another analogy for a class (as opposed to its instances) is that the class is like a **recipe** for how to make instances. The instances are like food cooked from the recipe (say, chocolate chip cookies). It isn't hard to tell the difference between these things. The cookies smell good. If you are hungry, the note−card with the recipe on it won't be very satisfying. (It probably tastes a lot like cardboard.) On the other hand, if you're going over to Grandma's to cook, you might want to take the recipe but you probably don't want to stick the chocolate chip cookie in your back pocket. Classes actually contain a lot of information other than just how to make an instance. (The recipe might, too. It might include information on how long it takes to make the cookies, whether they need to be refrigerated, how long it will take before they go stale, or even how many calories they contain.)

Figure 7.2. Two recipes (classes) and two platefuls of cookies (instances) made from the second recipe.

7.1.3 Classes are Types

Like interfaces, classes represent particular *kinds* of objects, or ***types*** . Once a class has been defined (see below), its name can be used to declare variables that hold objects of that type. So an instance of a class can be labeled using a name whose declared type is that class. For example, the CheckBoxes described above are labeled using names (*yesCheckBox* and *noCheckBox*) whose declared type is CheckBox. Note that the class CheckBox — the CheckBox recipe — can't be labeled using a name whose declared type is CheckBox. The type of the **class** CheckBox is *Class*, not *CheckBox*. (This is the recipe versus cookie distinction again.)

If an object is an instance of a class — such as *yesCheckBox* and the class CheckBox — then the type membership expression

```
yesCheckBox instanceof CheckBox
```

has the value `true`. Of course,

```
CheckBox instanceof CheckBox
```

is `false` (since the class isn't a CheckBox), but

```
CheckBox instanceof Class
```

is `true`.

Style Sidebar

Class Declaration

It is conventional to declare the members of a class in the following order:

- static final fields (i.e., constants)
- static non−final fields
- non−static fields
- constructors
- methods

This order is not necessary — any class member can refer to any other class member, even if it is declared later — but it makes your code easier to read and understand.

All non−private members of the class should be listed in the class's documentation.

7.2 Class Declaration

A ***class definition*** starts out looking just like an interface declaration, although it says that it is a class rather than an interface:

```
class Cat {
    ...
}
```

A class definition tells you what type of thing it is — a class — what it is called — its name — and what it's made of — its definition, between braces. This last part is called the class's ***body***. The body of the class definition contains all of the information about how instances of that class behave. It also gives instructions on how to create instances of the class. These elements — ***fields***, ***methods***, and ***constructors*** — are called the class's ***members***.

[Footnote: Be careful not to confuse members, which are parts of the class, with instances, which are objects made from the class. If chocolate chip cookies are instances of the cookie class (recipe), the chocolate chips are members of the class.]

Each member is declared inside the body of the class, but not inside any other structure within the class. Another way of saying this is that each member is declared at ***top level*** within the class. So members are all and only those things declared at top level within a class.

For example, each instance of Java's ***Rectangle*** class has a set of four coordinates describing the rectangle's position and extent, as well as methods including one which tells whether a particular (x,y) pair is *inside* the Rectangle.

```
class Rectangle {
    ...
    int height;
    int width;
    int x;
    int y;
    ...
    ...inside(...)...
}
```

In this case, *height*, *width*, *x*, *y*, and *inside* are all members of the Rectangle class.

Members and instances are quite different:

- **Members are parts of a class.**
- **Instances are things created from the class.**

We will return to each of the elements of this declaration later in this chapter.

7.2.1 Classes and Interfaces

A class may ***implement*** one or more interfaces. This means that the class subscribes to the promises made by those interfaces. Since an interface promises certain methods, a class implementing that interface will need to provide the methods specified by the interface. The methods of an interface are ***abstract*** — they have no bodies. Generally, a class implementing an interface will not only match the method specifications of the interface, it will also provide ***bodies — implementations*** — for its methods.

For example, a ScoreCounter class might meet the contract specified by the Counting interface:

```
interface Counting {
    void increment();
    int getValue();
}
```

So might a Stopwatch, although it might have a totally different internal representation. Both would have *increment* and *getValue* methods, but the bodies of these methods might look quite different. For example, a ScoreCounter for a basketball game might implement *increment* so that it counts by 2 points each time, while a Stopwatch might call its own *increment* method even if no one else does.

A class that implements a particular interface must declare this explicitly:

```
class ScoreCounter implements Counting {
    ...
}
```

If a class implements an interface, an instance of that class can also be treated as though its type were that interface. For example, it can be labeled with a name whose declared type is that interface. For example, an instance of class ScoreCounter can be labeled with a name of type Counting. It will also answer true when asked whether it's an `instanceof` that interface type: if *myScoreCounter* is a ScoreCounter, then

```
myScoreCounter instanceof Counting
```

is `true` . Similarly, you can pass or return a ScoreCounter whenever a Counting is required by a method signature.

The generality of interfaces and the inclusion of multiple implementations within a single (interface) type is an extremely powerful feature. For example, you can use a name of type Counting to label either an instance of ScoreCounter or an instance of Stopwatch (and use its *increment* and *getValue* methods) without even knowing which one you've got. **This is the power of interfaces!**

7.3 Data Members, or Fields

The Rectangle class, above, had certain things that were a part of each of its instances: *width*, *height*, etc. This is because part of what it is to be a Rectangle involves having these properties. A Rectangle−factory (or Rectangle−recipe) needs to include these things. Of course, each Rectangle made from this class will have its *own* width, height, etc. — it wouldn't do for every Rectangle to have the *same* width!

[Insert pic of rectangles]

Many objects have properties such as these: information called ***state*** or ***data*** that each instance of a class needs to keep track of. This kind of information is stored in parts of the object called fields. A ***field*** is simply a name that is a part of an object. For the most common kind of field, each instance of a class is born with its own copy of the field — its own label or dial, depending on the type of name the field is.

Declaring a field looks just like an ordinary name declaration or definition (depending on whether the field is explicitly initialized). Such a declaration is a field declaration if it takes place at ***top level*** in the class, i.e., if it is a class member. (A local variable declared inside a method body or other block is not at top level in the class.)

Consider the Rectangle class defined above and reproduced here:

```
class Rectangle {
    int height;
    int width;
    int x;
    int y;
    ...
}
```

Each instance of this class will have four `int`-sized dials associated with it, corresponding to the height, width, horizontal and vertical coordinates of the Rectangle instance. These fields are declared at top level inside the class body.

These fields are declared here, but not initialized: none of these fields is explicitly assigned a value. Fields, unlike variables, are initialized by default. If you don't give a field a value explicitly, it will have a default value determined by its type. For example, `int` fields have a default value of 0. Contrast `int` local variables, which don't have a default value and cannot be used until they are initialized. For details on the default values for each type, see the sidebar on Default Initialization.

Java Types and Default Initialization

In Java, field names can be declared without assigning them an initial value. In this case, Java automatically provides the field with a ***default value***. The value used by Java depends on the type of the field.

Fields with numeric types are initialized by default to the appropriate 0; that is, either `0` or `0.0` (using the appropriate number of bits).

Fields with type `char` default to the value of the character with ascii and unicode code 0 — `'\u000'`. This character is sometimes called ***the null character***, but should not be confused with the special Java value `null`, the non-pointer.

Fields with `boolean` type are by default assigned the value `false`.

Fields associated with label (reference) types — including *String* — are by default not bound to any object, i.e., their default value is `null`.

If a declaration is combined with an assignment — i.e., a definition — the definition value is used and these default rules do not apply.

These rules apply to names of fields as well as to the components of arrays — described in Chapter 12, *Dealing with Difference: Dispatch*. In contrast, local variables must be explicitly assigned values — either in their declaration (definition) or in a subsequent assignment statement — before they are used. There are also names called ***parameters***, which appear in methods and `catch` expressions; they are initialized by their invoking expressions and are discussed elsewhere in this book.

7.3.1 Fields are Not Variables

The difference in default initialization is only one difference between fields and local variables. This section covers several other important differences after first reviewing some properties of local variables.

A ***local variable*** is a name declared inside a method body. The ***scope of a local variable*** — the space within which its name has meaning — is only the enclosing block. At most, this is the enclosing method, so the maximum lifetime of a variable name is as long as the method is running. Once the method exits, the variable goes away. (A similar variable will come into existence the next time the method is invoked, but any information stored in the variable during the previous method invocation is lost.)

7.3.1.1 Hotel Rooms and Storage Rental

Because a field is a part of an object, and because an object continues to exist even when you're not explicitly manipulating it, fields provide longer−term (persistent) storage. When you exit a block, any variables declared within that block are cleared away. If you reenter that block at some later point, when you execute the declaration statement, you will get a brand new variable. This is something like visiting a hotel room. If I visit Austin frequently, I may stay in similar (or even the same) hotel rooms on each trip. But even if I stay in the same hotel room on subsequent visits, I can't leave something for myself there. Every time that I check into the hotel, I get what is for all intents and purposes a brand new room.

Contrast this with a long−term storage rental. If I rent long−term storage space, I can leave something there on one visit and retrieve it the next time that I return. Even if I leave the city and return again later, the storage locker is mine and what I leave there persists from one visit to the next. When I'm in Seattle, the things I left in my rental storage in Austin are still there. When I get back to Austin, I can go to my storage space and get the things I left there. This is just like a field: the object and its fields continue to exist even when your attention is (temporarily) elsewhere, i.e., even when none of the object's methods are being executed.

The storage locker story is actually somewhat more complex than that, and so is the field story. It might be useful for someone else to have a key to my storage locker, and it is possible for that person to go to Austin and change what's in the locker. So if I share this locker with someone else, what I leave there might not be what I find when I return. It is important to understand that this is still not the same as the hotel room. Between my visits, the hotel cleans out the room. If I leave something in my hotel room, it won't be there the next time I come back. Each time, my hotel room starts out “like new.” In contrast, the contents of my storage locker might change, but that is because my locker partner might change it, not because I get a freshly cleaned locker each time that I visit.

The locker partner story corresponds closely to something that can happen with fields. It is possible for the value of a field to change between invocations of the owning object's methods, essentially through the same mechanism (sharing) as the storage locker. To minimize this (when it is not desired), fields are typically declared `private`. For more on this matter, see the discussion of `public` and `private` in the next chapter. We will return to the issue of shared state (e.g. when two or more people have access to the same airport locker) in Chapter 20, *Synchronization*.

7.3.1.2 Whose Data Member Is It?

A second way in which fields differ from variables is that every field belongs to some object. For example, in the Rectangle code, there's no such thing as *width* in the abstract. Every *width* field belongs to some *particular* Rectangle instance, i.e., some object made from the Rectangle class/factory/recipe.

Because a field belongs to an object, it isn't really appropriate to refer to it without saying *whose* field you are referring to. Many times, this is easy:

```
myRectangle.width
```

for example, if you happen to have a Rectangle named *myRectangle*. The syntax for a field access expression is

1. An object−identifying expression (often, but not always, a name associated with the object), followed by
2. a period, followed by
3. the name of the field.

You can now use this as you would any other name:

```
myRectangle.width = myRectangle.width * 2;
```

for example.

There is, however, a common case in which the answer to the question "whose field is it?" may be an object whose name you don't know. This occurs when you are in a class definition and you want to refer to the instance whose code you are now writing. (Since a class is the set of instructions for how to create an instance, it is common to say "the way to do this is to use my own *width* field...")

In Java, the way to say "myself" is ***this***. That is, `this` is a special name expression that is always bound to the current object, the object inside whose code the name `this` appears. That means that the way to say "my own *width* field...." is **`this.width`**. (Note the period between *this* and *width* — it is important!)

7.3.1.3 Scoping of Fields

The final way in which fields differ from (local) variables is in their scoping. The scope of a name refers to the segment of code in which that name has meaning, i.e., is a legitimate dial or label. (If you refer to a name outside of its scope, your Java program will not compile because the compiler will not be able to figure out what you mean by that name.) A local variable only has scope from its declaration to the end of the enclosing block. (A method's parameter has scope throughout the body of that method.)

A field name has scope anywhere within the enclosing class body. That means that you can use the field name in any other field definition, method body, or constructor body throughout the class, including the part of the class body that is textually prior to the field declaration! For example, the following is legal, if lousy, Java code:

```
class Square {
    int height = this.width;
    int width = 100;
    ...
}
```

This isn't very good code because (a) it's convoluted and (b) it doesn't do what you think it does. Although *this.width* is a legal expression at the point where it's used, the value of *this.width* is not yet set to 100. The result of this code is to set *height* to 0 and *width* to 100. The rule is: all fields come into existence simultaneously, but their initialization is done in the order they appear in the class definition text.

A cleaner version of this code would say

```
class Square {
    int height = 100;
    int width = this.height;
    ...
}
```

A Comparison of Kinds of Names

	Scope	Lifetime	Default initialization
Class or Interface Name	Everywhere within containing program or package.	Until program execution completes.	—–
Field (Data Member)	Everywhere within containing class.	Lifetime of object whose field it is.	Label names: `null` Dial names: value depends on type.
Parameter	Everywhere within method body.	Until method invocation completes.	Value of matching argument expression supplied to method invocation.
(Local) Variable	From declaration to end of enclosing block.	Until enclosing block exits.	Illegal to use without explicit initialization (when declared or in a subsequent assignment).

[Footnote: The row for Class or Interface Name refers only to top−level (non−inner) classes or interfaces. The scope and lifetime of an inner class is determined by the context of its declaration. See Chapter 13, *Encapsulation*.]

7.3.2 Static Members

So far, we've said that fields belong to instances made from classes and that each instance made from the class gets its own copy. Recall that the class itself is an object, albeit a fairly different kind of object. (The class is like a factory or a recipe; it is an instance of the class called *Class*.) Sometimes, it is useful for the class object itself to have a field. For example, this field could keep track of how many instances of the class had been created. Every time a new instance was made, this field would be incremented. Such a field would certainly be a property of the class (i.e., of the factory), not of any particular instance of that class.

The declaration for a class object field looks almost like an instance field. The only difference is that class field declarations are preceded by the keyword ***static***.

[Footnote: The choice of the keyword `static`, while understandable in a historic context, strikes us as an unfortunate one as the common associations with the term don't really accord with its usage here. In Java, `static` means “belonging to the class object.”]

For example:

```
class Widget {
    static int numInstances = 0;
    ...
}
```

In this case, individual Widgets do *not* have *numInstances* fields. There is only one *numInstances* field, and it belongs to the factory, not the Widgets. To access it, you would say *Widget.numInstances*. In this case, *this.numInstances* is not legal code anywhere within the Widget class.

Style Sidebar

Field Documentation

In documenting a field, you need to indicate what that field represents conceptually to the object of which it is a part. In addition, you should answer these questions as appropriate:

- What range of values can this field take on?
- What other values are interdependent with this one? For example, must this field's value always be updated in concert with another field, or must its value remain somehow consistent with another field?
- Are there any “special” values of this field that carry hidden meaning?
- What methods (or constructors) modify this field? Which read this field? What else relies on its value?
- Where does the value of this field come from?
- Can the value of this field change?

7.4 Methods

In Chapter 4, *Specifying Behavior: Interfaces*, we saw how ***method signatures*** describe the name, parameters, and return type of a method. A method signature declared in an interface ends in a semi−colon; this method specifies a contract, but doesn't say anything about how it works. It is essentially a rule specification. This kind of method — a

specification without an implementation — is called ***abstract***.

Classes specify more than just a contract. Classes also specify how their instances work. In order for an instance to do be able to do something, its class must give more than the rule specification for its methods. An instance needs the rule body for its methods. Classes must supply bodies for any methods promised by the interfaces that they implement. They may also supply additional methods with their own signatures and bodies.

Methods can be identified by the fact that a method name is always followed by an open parenthesis. (There may then be some parameters, as discussed below; there will always be a matching close parenthesis as well.)

7.4.1 Method Declaration

A ***method definition*** also follows the *Type−of−thing Name−of−thing* convention, but the *type−of−thing* is the type that is *returned* when the method is called. So, for example, the *inside* method in the definition of Rectangle, described earlier in this chapter, returns a `boolean` value:

```
boolean inside(int x, int y) {
    ...
}
```

Inside the parentheses is the list of parameters to the method: calling *pictureFrame.inside* on a particular *x* and *y* value returns `true` or `false` depending on whether the point (*x*, *y*) is inside *pictureFrame*. For example,

```
pictureFrame.inside(120, 83)
```

returns `true` if (120, 83) is inside *pictureFrame*, and `false` otherwise.

Remember that the *inside* method only exists with reference to a particular Rectangle — it's always *some* object's method!

The list of parameters, like every other declaration, follows the *Type−of−thing Name−of−thing convention*. Note, though, that while a regular variable definition can declare multiple names with a single type, in a parameter list each name needs its own type.

A few more notes on methods: If there are no parameters, the method takes no arguments, but it must still be declared and invoked with parentheses:

```
pictureFrame.isEmpty()
```

for example. If there is no return value, the return type of the method is ***void***. Finally, inside the body of the method, the parameters may be referred to by the names they're given in the parameter declaration. It doesn't matter what other names they might have had outside of the method body, or what else those parameter names might refer to outside the method body. We'll return to the issue of scoping later.

Recall from previous chapters that the method definition as we've described it so far — the return type and the parameter list — is also called the ***signature*** of the method. It tells you what types of arguments need to be supplied when the method is called — it must be possible to assign a value of the argument type to a variable of the parameter type — and what type of thing will be returned when the method is invoked. It doesn't tell you much about the relationships between the method's inputs and its outputs, though. (The method's documentation ought to do that!)

Style Sidebar

Method Implementation Documentation

Documentation for methods in classes is much like the documentation for methods in interfaces. However, class/object methods have bodies as well as signatures. In addition to the usual documentation of the method signature (see the Style Sidebar on Method Documentation in Chapter 4, *Specifying Behavior: Interfaces*), your method documentation here should include:

- Ways in which this method implementation differs from or specializes the documented interface method (signature).
- Information concerning the design rationale (why the method works the way that it does), just as you would for any piece of Java code. For more detail, see the Style Sidebar on Documentation in Chapter 6, *Statements and Rules*.

7.4.2 Method Body and Behavior

This relationship — how to get from the information supplied as arguments to the result, or return value — is the “how to do it” part of the method. Its details are contained in the ***method body***, which — like a class body — goes between a pair of braces. What goes in here can be variable definitions or method invocations or any of the complex statements that you will learn about later. You cannot, however, declare other methods inside the body of a method. Instead, the method body simply contains a sequence of instructions that describe how to get from its inputs (if any) to its output (if any), or what else should happen in between.

The body of a method is inside the scope of its parameters. That is, the parameter names may be used anywhere within the method to refer to the corresponding arguments supplied at method invocation time. The body of an instance method is also within the scope of the special name ***this***. Just as in fields, inside a method the name `this` refers to the particular instance whose method this is. Static methods — methods belonging to the class, as discussed in the next subsection — are not within the scope of `this`, though. That is, you can't use the special name expression `this` in a static method.

In order to return a value from a method, you use a special statement: ***return***. There are actually two forms of this statement:

```
return (...);
```

returns a value (whatever is in the parentheses) from a method invocation. For example,

```
return (total + 1);
```

returns one more than the value of *total*, though it doesn't change the value of *total* at all. The parentheses around the expression whose value is to be returned are in fact optional, leading to the second form of return:

```
return;
```

is used to exit from a method whose return type is `void`, i.e., that does not return anything.

Remember (from Chapter 5, *Expressions: Doing Things with Things*) that a ***method invocation*** is an expression whose type is the return type of the method and whose value is the value returned by the method. You make this happen (when you're describing the method rule) by using an explicit `return` statement in a method's body. In Chapter 6, *Statements and Rules*, we saw the execution rule for a method body and how it relates to the evaluation rule for method invocation. This process is summarized in the sidebar on Method Invocation and Execution at the end of this section.

7.4.3 A Method ALWAYS Belongs to an Object

A method is a thing that can be *done* (or ***invoked***, or ***called***). For example, a painting program can draw a line, so *drawLine* could be the name of a method. **Every method belongs to a particular object.** For instance, each *getValue* method belongs to a particular ScoreCounter (or Stopwatch, or...) object; there is no such thing as an independent *getValue* method. So, if *myScoreCounter* refers to a particular ScoreCounter,

```
myScoreCounter.getValue()
```

invokes *myScoreCounter*'s `int`-returning method. You can't just call *getValue*. Whose *getValue* method is it, anyway?

Each time that you refer to a method, you should ask yourself whose method it is. You can invoke a method by first referring to the object, then typing a period, then the method name, as in

```
myScoreCounter.getValue()
```

Sometimes, the answer to "whose method is it?" will be "my own", that is, the method belongs to the object whose code is being executed. As with fields, the way to say "myself" is with the special name expression ***this***, so the way to say "my *getValue* method" is

```
this.getValue()
```

Note the period between *this* and *getValue()* — it is important!

Generally, methods belong to instances of the class in which they're defined. Occasionally, though, it may be useful to have a method that belongs to the class itself. This corresponds to a property of the factory (or recipe), rather than one belonging to the widgets (or cookies) produced. For example, a method that prints out the number of widgets produced by the factory so far would be a method belonging to the factory, not one belonging to any particular widget. Methods that belong to the class instead of to its instances look just like regular methods, except that they are prefaced with the keyword ***static***. (This name is pretty unintuitive, though it makes some sense in its historical context. Remember: In Java, `static` means "belonging to the class/factory/recipe itself, not to its instances.") A static method can be addressed by first citing the object it belongs to, then period, then the method name, for example:

```
Widget.howManyWidgets()
```

A static method *should not* be invoked using keyword `this`, though, because it doesn't belong to an instance. It is also possible to use keyword `this` *by itself* to refer directly to the object whose method the name `this` appears in. For example:

```
JButton button = new JButton("Press me");
button.addActionListener(this);
```

declares and constructs a new JButton object and then says that "this object" (the one that created the button) should listen for and respond to the event in which the user presses the button.

7.4.4 Method Overloading

Just as in an interface, it is possible for a class to have multiple methods with the same name. This is called ***method overloading***, since the name of the method is overloaded — it actually refers to two or more distinct methods — belonging to that object. In this case, each method must have a different ***footprint***, i.e., the ordered list of parameter types must

differ for two methods of the same object with the same name.

When an object has an overloaded method, the particular method to be invoked is selected by comparing the types of the arguments supplied with the footprints of the methods. The method whose footprints best matches the (declared) types of the arguments supplied is the one that is invoked. This matching is done using the same type inclusion rules as the operator `instanceof`.

Method Invocation and Execution

Method invocation is an expression; it is evaluated, producing a value. Within this expression, the body of the method is treated as a block (sequence) statement to be executed. This sidebar summarizes this process.

1. Before the method invocation expression can be evaluated, the object expression describing whose method it is must be evaluated. This object is called the method's ***target***.

2. Based on this object and the (declared) types of the argument expressions, the method body is selected.

3. The argument expressions are evaluated and the method parameter names are bound to the corresponding arguments. If the target is an instance (i.e., if the method is not static), the name `this` is bound to the target as well.

4. Within the scope of these name bindings, the body of the statement is executed as a normal block except for special rules concerning `return` statements.

    - If, at any point within the execution of the body, a `return` statement is encountered, its expression (if present) is evaluated and then the entire method body and the scope of parameter names and `this` are exited upon completion of the `return` statement.

    - If the method has a return type other than `void`, the `return` statement is mandatory and must include an expression whose type is consistent with the return type. A suitable `return` statement must be encountered on any normal execution path through the method body. In this case, the value of the `return` expression is the value returned by the method invocation expression.

    - If the return type of the method is `void`, the final closing brace of the method body is treated as an implicit

      ```
      return;
      ```

      statement, i.e., a `return` with no expression. This has the effect of exiting the method body and special name scope.

7.5 Constructors

So, how do objects get created? Each class has a special member, called a ***constructor***, which gives the instructions needed to create a new instance of the class. (If you don't give your class a constructor, Java automatically uses a default constructor, which roughly speaking “just creates the instance” — details below. So some of the classes that you see may not appear to have constructors — but they all do.)

7.5.1 Constructors are Not Methods

A constructor is sort of like a method.

1. It has a (possibly empty) parameter list enclosed in parentheses.
2. It has a body, enclosed in braces, consisting of statements to be executed.
3. Inside the constructor body, `this.` expressions can be used to refer to methods and fields of the individual instance under construction.

There are several differences.

1. The name of a constructor always matches the name of the class whose instances it constructs.
2. A constructor has no return type.
3. A constructor does not return anything; `return` statements are not permitted in constructors.
4. A constructor cannot be invoked directly.

Instead, a constructor is invoked as a part of a `new` expression. The result of evaluating this `new` expression is a new instance of the type whose constructor is evoked.

For example, class Pie might have a constructor as follows:

```
class Pie {

    Pie(Ingredients stuff) {
        stuff.bake();
    }
}
```

In other words, to create a Pie, bake its ingredients. Note that *stuff* is a parameter, just like in a method. Constructor parameters work exactly like method parameters, and constructors take arguments to match these parameters in the same way that methods take parameters.

But you don't invoke a constructor in the same way that you invoke a method. In order to invoke a method, you need to know whose method it is. In order to use a constructor, you only need to know the name (and parameter type list) of the constructor. You invoke a constructor with a `new` expression as follows:

```
new Pie(myIngredients)
```

where *myIngredients* is of type Ingredients.

7.5.2 Syntax

The syntax of a constructor is similar to, but not identical to, the syntax of a method. A constructor may begin with a visibility modifier (i.e., `public`, `protected`, or `private` — see Chapter 8, *Designing with Objects*, for details) or one of a handful of other modifiers. Next comes the name of the constructor, which is always identical to the name of the class. The name is followed by a comma−separated parameter list enclosed in parentheses. This parameter list, like the parameter list of a method, consists of *Type−of−thing Name−of−thing* pairs. As in a method, the constructor name plus the ordered list of parameter types forms the constructor's footprint. It is possible for a class to have multiple constructors as long as they have distinct footprints.

After the parameter list, a constructor has a body enclosed in braces. This body is identical to a method body — an arbitrary sequence of statements — except that it may not contain a `return` statement. This is because constructors are not methods that can be called and that return values of specified types; instead, a constructor is invoked using a `new` expression whose value is a new instance of the constructor class's type. The constructor body may contain any other kind of expression or statement, however, including declarations or definitions of local variables.

```
modifiers ClassName (type_1 name_1, ... type_n name_n) {
    // body statements go here
}
```

For example, the NameDropper class might begin as follows. Note that the constructor argument is used to initialize the private field, the particular name that *this* NameDropper will drop.

```
public class NameDropper extends StringTransformer {

    private String who;

    public NameDropper(String name) {
        this.who = name;
    }

    ...
}
```

Note the use of a `this.` expression to refer to the field of the particular NameDropper instance being created. Also note that the class and its constructor are `public`, while the field is `private`. We will see in Chapter 8, *Designing with Objects*, that this is a typical style.

This constructor could be invoked using the expression

```
new NameDropper("George")
```

or

```
new NameDropper("Lois")
```

Figure 7.3. Two NameDroppers, each asked to *transform* the String `"Hello"`.

Style Sidebar

Constructor Documentation

Although a constructor is not a method, documentation for a constructor is almost identical to documentation for a method. Constructor documentation should include:

- Specifics distinguishing this constructor from others.
- Preconditions for using this constructor.
- Parameters required and their role(s).
- Relationship of the constructed object to parameters or other factors.
- Side effects of the constructor.
- Additional assumptions and design rationale as appropriate.

7.5.3 Execution Sequence

Before a constructor is invoked, the instance is actually created. In particular, any dials or labels declared as fields of the instance are created before the execution of any constructor code. This permits access to these fields from within the constructor body. In addition, any initialization of these fields — through definitions in their declarations — is executed at this time as well. Fields are each created and then each initialized in textual order, but all fields — even those declared after the constructor — are created and initialized prior to the execution of the constructor.

[Footnote: There should be no such fields, declared after the constructor, because this makes your code difficult to read and so is bad style. However, if any such declarations are made, they still are executed prior to the constructor itself.]

Once each of the instance fields is created, execution of the constructor itself can begin.

When a constructor is executed, its parameters are matched with the arguments supplied in the invocation (`new`) expression. For example, in the body of the NameDropper constructor, the name *name* is identified with the particular String supplied to the constructor invocation expression. So if the constructor were invoked with the expression

```
new NameDropper("Terry")
```

the name *name* would be associated with the String `"Terry"` during the execution of the body of the NameDropper constructor. When the statement

```
this.who = name;
```

is executed, the value of the expression *name* is the String `"Terry"`.

Once each of the parameter names has been associated with the corresponding argument, the execution of the statements constituting the constructor's body proceeds in order (except where that order is modified by control−flow expressions such as `if` or `while`). These statements may include local variable declarations; in this case, the name declared has scope from its declaration to the end of the enclosing block, just as in a method. When the end of the constructor is reached, execution of the constructor invocation expression is complete and the value — the new instance — is produced.

Because a constructor body may not contain a `return` statement, it is not possible to exit normally from any part of the constructor body except the end. Judicious use of conditionals can simulate this effect, however.

7.5.4 Multiple Constructors and the Implicit No−Argument Constructor

A class may have more than one constructor as long as each constructor has a different footprint, i.e., as long as they have different ordered lists of parameter types. So, for example, NameDropper might also have a variant constructor that took a descriptive phrase as well as name:

```
public NameDropper(String name, String adjective) {
    this.who = adjective + " " + name;
}
```

In this case,

```
new NameDropper("Marilyn Monroe")
```

would create a NameDropper that started every phrase with *"Marilyn Monroe says..."* while

```
new NameDropper("Norma Jean", "My dear friend")
```

(i.e., **`NameDropper(String, String)`**) would attribute everything to *"My dear friend Norma Jean..."*

If — and only if — a class contains no constructors at all, a default constructor is assumed present. This default constructor takes no arguments and does nothing beyond creating the object (and initializing the fields if they are defined in their declarations).

If there is even one constructor, the implicit no−argument constructor is not assumed. This means that if you define a constructor such as the one for NameDropper, above, that takes a parameter, the class will not have a no−argument constructor (unless you define one).

Beware: This can cause a problem when extending a class, if you're not careful. See Chapter 10, *Inheritance*.

7.5.5 Constructor Functions

Often, one of the main functions of a constructor is to initialize the state of the instance you're creating. Some initializations don't require a constructor; they can happen when the field is declared, by using a definition instead of a simple declaration:

```
public class LightSwitch {
    private boolean isOn = false;
}
```

In this case, each LightSwitch instance is created in the off position. In this kind of initialization, each instance of the class has its field created with the same initial value.

Contrast this with the following example, in which the initial value of the *name* field isn't known until the particular Student instance is created.

```
public class Student {

    private String name;

    public Student(String who) {
        this.name = who;
    }
}
```

In this case, a constructor is used to initialize the field named *name* explicitly. When the initial value of a field varies from instance to instance, it cannot be assigned in the field declaration. Instead, it must be assigned at the time that the particular instance is created: in the constructor.

A constructor (or a method body) can also refer to properties of the class object itself. Recall the Widget class, which kept track of how many instances had been created. When the constructor is invoked, it can increment the appropriate field:

```
public class Widget {

    private static int numInstances = 0;

    public Widget() {
        Widget.numInstances = Widget.numInstances + 1;
    }

    public static int howManyWidgets() {
        return Widget.numInstances;
    }
}
```

Note that the constructor is not declared `static` — constructors don't properly belong to any object — but that it refers to a `static` field. Note also that the `static` field is referred to using the class name (**`Widget`**), *not* using **`this`**. We've also filled in the static method referred to previously.

Finally, note that there is no explicit `return` statement in a constructor. A constructor is not a method, and it cannot be invoked directly. Instead, it is used in a construction expression, with the keyword `new`:

```
new Widget()
```

is an expression whose type is Widget and whose value is a brand new instance of the Widget class, for example.

Question: Implement a Counter class which supports an *increment* (increase−by−one) method. Where does the Counter's initial value come from?

Style Sidebar

Capitalization Conventions

By convention, the first letters of all class and interface names are capitalized. Since constructor names match their classes, constructor names also begin with capital letters. Java file names also match the class (or interface) declared within, so Java file names begin with a capital letter.

All other names (except constants) begin with lower case letters. In particular, the names of Java primitive types begin with lower case letters, as do fields, methods, variables, and parameters.

After the first letter, mixed case is used, with subsequent capital letters indicating the beginnings of intermediate words: e.g., *ClassName* and *instanceName*.

The exception to the above conventions is the capitalization of constants (i.e., static final fields; see Chapter 8, *Designing with Objects*). The names of constants are entirely capitalized. Intermediate words are separated using underscores (_): *CONSTANT_NAME*.

Chapter Summary

- A Java class is a Java type.
- Each (public, top level) class must be defined in a separate file whose name matches the class name.
- An instance of a class is an object whose type is that class.
- If a class implements an interface, its instances must satisfy the interface's promises.
- Classes have methods, fields, and constructors.
- In a class, methods typically have bodies specifying how to carry out the method. (Otherwise, the method is `abstract`, and so is the class.)
- Every method belongs to some object. Unless declared `static`, a method belongs to (each of) a class's instances, not to the class itself.
- A field declares (and perhaps also defines) a name whose scope is the class body (i.e., any methods, fields, or constructors in the class body) and whose lifetime is the lifetime of the instance it belongs to.
- Every field belongs to some object. Unless declared `static`, a field belongs to (each of) a class's instances. Each instance has its own copy of the field, i.e., its own unique dial or label with that field's name and type.
- In Java, `this` is a special name, bound in any non−static member, that refers to the instance whose instructions are being followed. An instance can refer to its own methods and fields by saying **`this.`*`methodName`*`(...)`** or **`this.`*`fieldName`***, or to itself by the name expression **`this`**.
- A constructor gives instructions for how to create an instance of the class.
- The class itself is an object. (It is an instance of the class *Class*.) Fields and methods declared `static` belong to the class object itself and are properly referred to using **`ClassName.`*`methodName`*`(...)`** or **`ClassName.fieldName`**.

Exercises

1. Consider the following definition:

```
public class MeeterGreeter {

    private String greeterName;

    public MeeterGreeter(String name) {
        this.greeterName = name;
    }

    public void sayHello() {
        Console.println("Hello, I'm " + this.greeterName);
    }

    public void sayHello(String toWhom) {
        Console.println("Hello, " + toWhom
                        + ", I'm " + this.greeterName);
    }

    public String getNameWithIntroduction(String toWhom) {
        //****
        this.sayHello(toWhom);
        return this.greeterName;
    }
}
```

Now assume that the following definitions are executed:

```
MeeterGreeter pat = new MeeterGreeter("Pat");
MeeterGreeter terry = new MeeterGreeter("Terry");
```

a. What is printed by

```
pat.sayHello()
```

What is returned? Which method is invoked?

b. What is printed by

```
new MeeterGreeter("Chris").sayHello("Terry")
```

What is returned? Which method is invoked?

c. What is printed by

```
terry.sayHello("Pat")
```

What is returned? Which method is invoked?

d. Assume that the expression

```
pat.getNameWithIntroduction("Chris")
```

is being evaluated. What would the value be of each of the following expressions if they were to appear at the place where the comment `//****` appears?

i. **`toWhom`**

ii. **`this.greeterName`**

iii. **`name`**

iv. **`this.sayHello()`**

v. **`new MeeterGreeter("Pat")`**

vi. **`this.getNameWithIntroduction(toWhom)`**

2. Now consider the following modification of the MeeterGreeter code. Assume that we add the field definition

```
private static String greeting = "Hello";
```

We will want to make several other modifications to the MeeterGreeter code, as follows.

a. Write a *changeGreeting* method that allows a user to change the greeting string.
   i. What parameters should this take?
   ii. What should it return?
   iii. What should its body say?
   iv. To which object should this method belong?

b. Write an expression that invokes the *changeGreeting* method that you have written.

c. Next, modify the *sayHello* methods to replace the fixed string "Hello" with a reference to the greeting field. Whose greeting field is it?

3. Define a class whose instances each have one method, *rememberAndReturnPrevious*, that takes a String and returns the String it was previously given. Supply the first return value through the instance creation expression. Give an example of your code in use.

Part 3

Refining Designs

Chapter 8

Designing with Objects

Chapter Overview

- How do I design using objects and entities?

In the preceding chapters, we have seen how interfaces specify contracts and how classes implement them. We have used expressions and statements to create instructions that describe the processes of performing actions, making up method and constructor bodies. And we have used names to retain an object's state even while none of the object's methods is executing. In this chapter, we turn to the question of how we ***design*** systems using these various tools.

The first part of this chapter looks at one simple example to illustrate how the fields and methods of an object can be identified and implemented. Although the example is small, the principles described here are general and will be used in the design of any object−oriented program. This example also provides an opportunity to look briefly at the question of ***privacy***, or how an object separates internal information from information that it makes available to other objects.

The next section of this chapter turns to look at three important kinds of objects that appear in many systems. These kinds of objects — ***data repositories***, ***resource libraries***, and ***traditional objects*** — each play distinctly different roles in any system, and their designs reflect these roles. A fourth distinct kind of object — ***animate objects*** — is the topic of the next chapter.

The chapter concludes with a discussion of the ways in which different objects and types are interrelated.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Objectives of this Chapter

1. To become familiar with the identification of objects, methods, fields, interfaces, and classes from a problem description.

2. To recognize common kinds of objects and the roles that they play.

3. To learn to identify opportunities to use these patterns in designing systems.

8.1 Object Oriented Design

So far, you've seen a lot of Java how−to: how to declare, define, assign, and invoke variables of primitive and object types, classes, object instances, methods, and control flow. Now that you have some fluency with the basic building blocks of Java, it is time to start looking at why each of these constructs is used and how they are combined to build powerful programs. In this chapter, we'll look at objects and classes; in the next, we'll continue this discussion by focusing on instruction−followers and self−animating objects.

8.1.1 Objects are Nouns

When you are constructing a computational system, you need to build pieces of code to play various roles in the system you're constructing. To a first approximation, you can do this by writing down a description in English of the system and the interactions you want to have with it (and that you want its parts to have with one another), then mapping these things onto elements of Java. When you do this, you will find that Java objects correspond roughly to the ***nouns*** of your description.

To be a bit more precise, Java objects are things in your computational world, but not all of the things are Java objects. Some of the things will have primitive types — numbers, for example, will probably be `doubles` or `ints` — but most of the things that are important enough to represent and complex enough that Java doesn't have a built−in type for them will be objects in your world. This means that you will have to define a Java class which describes what this type of object is (more below).

For example:

- A counter has a number associated with it. When it starts out, the number is 0. You can increment the counter, and each time you do so, the number goes up by one. At any time, the counter can also be asked to provide the current value of its associated number.

The nouns in this paragraph are *counter* and *number*. (And *you*, but we'll assume that *you* either refers to the user, which we don't need to implement, or to some other component outside of the current system.) The *counter* will be a Java object; we can use an `int` for the *number* since it isn't asked to do anything, just to be there.

8.1.2 Methods are Verbs

When you write down your description, you will also find that there are lots of things that these objects do to/with/for one another, or that you want to do to/with/for them. These things correspond to the ***verbs*** in your English description, and they are the methods of your Java objects. Every verb has a noun associated with it — its subject — and **every Java method belongs to some object**.

In our basic counter example, the verbs are *increment* (and its alternate form, *goes up by one*) and *provide* (as in “provide the current value”). *Increment* is something you need to be able to do to the counter object. We could handle provide in either of two ways: we could give the counter someone or something to provide the value to, or we could ask it. We will adopt the second of these options, though we will return to the first option in the chapter on Communication Patterns. This means that the counter object is going to have to have (at least) these methods.

8.1.3 Interfaces are Adjectives

Interfaces and classes are both types. How do you know when to use which one? As a general rule of thumb, names (including parameters, fields, and local variables) should generally be declared using an interface type whenever possible. Constructor expressions, of course, require a class type.

Interfaces are good at capturing commonality. It is almost always useful to define an interface corresponding to the set of features of your objects that would hold for any implementation of them. For example, the need for any counter to have an *increment* method and a *getValue* method makes these good properties to encapsulate in an interface. No matter how we implement the counter, these method properties will hold. In contrast, the fact that most counters will keep track of their value using a field (perhaps even an `int` field is an implementation−specific detail that cannot be expressed in an interface. An interface talks about what an object can do, not about how it accomplishes these tasks.

A Counting interface might say:

```
interface Counting {
    void increment();
    int getValue();
}
```

Why would we use this? By referring to any actual counters by their interface type — Counting — rather than their implementation types, we make it possible for the implementor to modify details of the implementation — or, even, to change which underlying implementation we're using — without changing the code that uses it. We also avoid committing to any specific aspects of the implementation — such as the representation of the current value through a `long` or a `double` or even a String — that really shouldn't matter to the user of the class.

The name of this interface is only moderately adjectival, but most interfaces are named using ***adjectives***. For example, we have seen Resettable and will soon see Animatable, Runnable, and Cloneable. We could almost call Counting ***Incrementable*** instead.

8.1.4 Classes are Object Factories

So if the nouns are objects, the verbs are methods, and the interfaces are adjectives, what is left for the classes? Java classes are *kinds* of objects. They correspond, roughly, to machines (or ***factories***) that tell you (or Java) how to make new objects, not (necessarily) to anything explicitly in your English description.

For example, the class BasicCounter is something that tells Java how to make a new BasicCounter. It doesn't appear explicitly in the English description, but parts of the description are about it and other parts imply things about what it must say. The phrase “When it starts out, the number is 0” talks about ***initial conditions*** for BasicCounter objects; the class is the thing responsible for establishing these (since it is the factory where Counters are made).

For that matter, the class is responsible for establishing what the parts of an object are. “Parts” here refers to ***methods*** and ***fields***. What are the pieces of a BasicCounter object? In this case, its number (and maybe an associated display). What are the things a BasicCounter can do (or that we can do with/to a BasicCounter)? Increment and provide its value, at least. So the BasicCounter will most likely include a number field (which is going to be of type `int`), as well as methods corresponding to incrementing and value-providing. It will also initialize the number field to 0.

Question: Is this a static or dynamic initialization? Where does it take place?

Picture of counters

Figure 8.1. Counters.

Style Sidebar

Class and Member Documentation

This list summarizes many of the main features that good documentation will capture about classes and their members. For more detail, see the specific documentation sidebars in the previous chapter.

- Methods
  - parameters: type and role
  - return value: type and role
  - function: why you'd do it
  - “side effects”: what else it does (esp. values changed)
- Fields
  - type and role
  - how it changes & which methods use/change it
  - constraints and interdependencies
- Constructors
  - parameters: type and role
  - relation of parameters to the particular instance produced
  - “side effects”: what else it does (esp. values changed)
- Class
  - its interface, especially key methods and fields & how they interact

8.1.5 Some Counter Code

Here is a very basic implementation of the counter class:

```
class BasicCounter implements Counting {
    int currentValue = 0;

    void increment() {
        this.currentValue = this.currentValue + 1;
    }

    int getValue() {
        return this.currentValue;
    }
}
```

Some notes on this code:

- The class is a factory for making BasicCounters. Its body talks about what each individual BasicCounter looks like, not about the factory itself.

- Each individual BasicCounter has its **own** *currentValue* field. Each one starts out with the value 0, but they can change independently: each *currentValue* field belongs to a specific BasicCounter.

- We haven't included a constructor because, in this case, Java's default constructor does what we want. This is in general true when there is no dynamic initialization (each instance starts out in the same state).

- The *increment* and *getValue* methods are methods that belong to each BasicCounter instance. In each case, they refer to the *currentValue* field of *that* BasicCounter instance. We note this by using the java keyword ***this***.

Someone wanting to use a BasicCounter could now do so by invoking an instance creation expression with this BasicCounter factory:

```
new BasicCounter()
```

This expression is probably more useful if we embed it inside another expression or statement, e.g.,

```
Counting myCounter = new BasicCounter();
```

Note the use of the interface type when declaring the name, but the class type within the construction expression.

Now we can ask *myCounter* to increment itself or to give us its value:

```
myCounter.increment();
Console.println(myCounter.getValue()); // prints 1
myCounter.increment();
myCounter.increment();
myCounter.increment();
Console.println(myCounter.getValue()); // prints 4
```

Final

A name in Java may be declared with the modifier ***final***. This means that the value of that name, once assigned, cannot be changed. Such a name is, in effect, ***constant***.

The most common use of this feature is in declaring ***final fields***. These are object properties that represent ***constant values***. Often, these field are static as well as final, i.e., they belong to the type object rather than to its instances. Making a constant static as well as final makes it easy for other objects to refer to this value. It is appropriate for static final fields to be declared `public` and to be accessed directly by other objects. Static final fields are the only fields allowed in interfaces.

In addition to final fields, Java parameters and even local variables can be declared final. A ***final parameter*** is one whose value may not be changed during the execution of the method. A ***final variable*** is one whose value is unchanged during its scope, i.e., until the end of the enclosing block.

[Footnote: Final fields and parameters are unnecessary unless you plan to use inner classes. They may, however, allow additional efficiencies for the compiler, and in any case they cannot be detrimental.]

Java methods may also be declared ***final***. In this case, the method cannot be overridden in a subclass. Such methods can be inlined by the compiler, i.e., the compiler can make these methods execute more efficiently than other non−final methods. A static method is implicitly final. An abstract method may not be declared final.

Java classes declared final cannot be extended (or subclassed).

8.1.6 Public and Private

When we defined the BasicCounter class, we intended that the rest of the world would interact with its instances (things produced by the BasicCounter factory) only through

increment and *getValue*. But there is nothing about the code we've written that prevents someone from defining a BasicCounter name and then changing the value of that BasicCounter instance's *currentValue* field. For example, it would be perfectly possible for another object to say

```
BasicCounter anotherCounter = new BasicCounter();
anotherCounter.currentValue = anotherCounter.currentValue + 1;
```

instead of

```
anotherCounter.increment();
```

This would be rather rude of it (and very bad style), but it is technically possible and unfortunately done all of the time. Using the interface type — Counting — rather than the class type — BasicCounter — is one way to avoid this, and this is yet another reason why it is generally better to use the interface type. But as the implementor of BasicCounter, we can't require that it always be treated as a Counting instead of as a BasicCounter. Further, ***coercion*** (such as **`(BasicCounter) myCounter`**) will get you around the interface−associated name.

[Footnote: Specifically, it would be legal, if longwinded, to say

```
((BasicCounter) myCounter).currentValue =
          ((BasicCounter) myCounter).currentValue + 1;
```

]

Class designers don't always get to choose how users of the class will interact with it or as what type they'll choose to treat it.

We can take a stronger position on the matter of direct field access, though. We can, in fact, prevent direct field access by ***protecting*** the *currentValue* field of each BasicCounter instance. We do this by changing the declaration of the field in BasicCounter:

```
class BasicCounter {
    private int currentValue = 0;

    void increment ...
}
```

By making *currentValue* ***private*** to class BasicCounter, only the instance of BasicCounter itself can access the *currentValue* field. Now, this rudeness on the part of the calling object would simply be impossible. (The compiler would complain that the calling object could not access BasicCounter's private field *currentValue*.)

In general, it's a good idea to define fields as private when you don't want them to be accessed directly by other objects. You can also define ***private methods***, which are generally things an object uses for its internal computations but not intended to be used from outside the object. Private things are a part of the class's or its instances' own internal representations and machinations; they are not to be shared.

Any member, not just a field or a method, can be private. You can even define ***private constructors***. Although this may seem like an odd thing to do, it actually isn't all that strange. It means that the class object (along with any instances it creates) maintains complete control over whether and when new instances can be created. The class can refuse to create any instances, or it can create just one instance and return this any time someone asks for a new one (using a special method the class defines for this purpose, such as *getInstance*, not the (private) constructor), or it can ask for the secret password before creating an instance if it (or its designer) wants to.

The opposite of *private* is ***public***. You should declare things `public` when you want them to be accessible from any part of anyone's code. You can also declare classes and interfaces to be public, in which case they must be defined in a file whose name is the same as the name of the class or interface, plus *.java*.

If you don't declare something private or public, it is in an intermediate state. There are actually two intermediate states, ***protected*** and the default state. These two are in fact equivalent to one another and to public unless you use *packages*, a Java feature that we will explore in the chapter on Abstraction. Until then — until you are building complex enough code that you need to subdivide it at finer levels than all−or−none — you should use public and private all of the time, i.e., everything in your code should be one or the other.

8.2 Kinds of Objects

Objects are the nouns of programming: the people, places, and things. Nouns do a lot of different things in the world and, similarly, objects do a lot of different things in programs. In this section, we take a closer look at several kinds of objects, their typical construction, and why you might use them. The objects discussed here are all relatively passive; they do nothing until asked. In the next chapter, we go on to look at active objects, objects that have their own instruction followers.

8.2.1 Data Repositories

A ***data repository*** is a very simple object that exists solely to hold a set of interrelated data. The data repository object simply glues these things together, providing a convenient way to deal with the grouped data as a single unit.

One example of a data object might be a postal address. This might consist of a street address, a city or town, a state or province, a postal code, and a country. There isn't really

much that you would do with an address, other than pull out the individual pieces or maybe modify one or more of the pieces. (For example, the postal service just changed my postal code, so although my address object stayed the same, its postal code field needed to change.) The whole address is useful and meaningful in a way that the pieces individually are not, so it is often convenient to be able to package these pieces together and to pass the address object around as a single unit.

Picture of address object

Figure 8.2. An *Address* object.

Here is some code for a very simple address object. Note that this code has some aesthetic problems, which we will address shortly.

```
public class OversimplifiedAddress {
    public String streetAddress,
                  city,
                  state,
                  postalCode;
    // Problems with this class:
    //   - Non-final fields ought not to be public.
    //   - Fields ought to be initialized
    //      by (missing) constructor or default.
}
```

Like instances of this OversimplifiedAddress class, data repository objects exist to hold a collection of pieces together. Typically, each of these pieces is represented by a field of the object. The simplest form of data repository object is one — like an instance of the Oversimplified Address class — that has a set of public fields and nothing else. However, this form is not recommended.

One object should never access another object's fields directly. Instead, an object should provide methods for accessing its fields.

[Footnote: Actually, this should read “One object should never access another object's *non−final* fields directly.” Final fields are in effect constants; the reasons for objecting to field access do not apply to read−only accesses to a constant.

Also, if you are concerned about the overhead of invoking a method: Where “getter” methods are simply long−winded ways of doing field access, a good compiler should be able to ***inline*** this code, which means that it replaces the method invocation with direct access to the field data when the code is translated to machine code. In Java, such inlining can be demanded by declaring the getter method to be `final`.]

In our simple address object, we violated this rule. To fix that class definition, we should instead make each of these fields internal to the object. So that other objects can access these fields, we need to provide *getter* and *setter* methods to access them. A ***getter*** method is a method that returns the value of a field. A ***setter*** method is one that has a single parameter, the new (desired) value of the field; evaluating this method modifies the state of the object to reflect this new value. Getter methods are sometimes called ***selectors*** and setter methods are sometimes called ***mutators***. It is common to use the name of the field prefixed with *get* as the name of the getter method and the name of the field prefixed with *set* as the name of the setter method.

Note that getter and setter methods need not correspond one to one with fields. Instead, a setter method may change the value of more than one field; a getter value may return an object that encapsulates more than one field value. Alternately, a getter or setter may make reference to an apparent field that doesn't actually exist *per se*.

We can improve the address class by modifying it to use getter and setter methods. Only one pair of these methods is shown here, although the complete class definition would presumably contain four pairs of getter and setter methods.

```
public class BetterAddress {
    private String streetAddress,
             city,
             state,
             postalCode;
    ...

    public void setPostalCode(String code) {
        this.postalCode = code;
    }

    public String getPostalCode() {
        return this.postalCode;
    }
    // Remaining problems with this class:
    //   - Fields ought to be initialized
    //      by (missing) constructor or default.
}
```

Why shouldn't one object access the fields of another directly? (Why should you use getter and setter methods?)

1. Methods separate use from actual (internal) representation. The user of a class shouldn't need to know (or care) how information is actually represented inside the class. For example, US postal codes are commonly written as five−digit numbers. A different implementation of addresses intended for use only in the US might actually represent the postalCode field using an `int` instead of a String. The getter and setter methods of this USAddress object could do the conversion for the user:

   ```
   public String getPostalCode() {
       return new String( this.postalCode );
   }
   ```

   We might have an interface (say, GeneralizedAddress) containing (an abstract version of) this method. Both USAddress and BetterAddress classes could implement the GeneralizedAddress interface, even though they use different internal representations.

   Another variant of separating use from actual representation involves getter and/or setter methods for fields that don't actually exist. For example, it might be useful for these address objects to have a getAddressLabel field, which would return the multiline String containing the complete address suitable for printing on an envelope. This getter method would automatically calculate the appropriate value from the individual fields of the address object; there is no actual field corresponding to the information that this getter field provides.

```
public String getAddressLabel() {
    return new String(this.streetAddress + "\n"
                         + this.city + ", "
                         + this.state + "  "
                         + this.postalCode + "\n"
                         + this.country );
}
```

Getter and/or setter methods like this one, which do not correspond to any actual field of the object, are sometimes called ***virtual fields***. To the user of the object, it looks as though there's a field there. Whether that field actually exists or just looks like it is nobody's business but the implementing object's.

2. Methods can provide additional behavior, including access control and error checking. For example, BetterAddress could be augmented with an internal list of the states or provinces within each country. If the setter method were given an argument that didn't match one of the appropriate values, it could report an error. The most extreme case of this is a read−only field, one in which no non−private setter method is supplied. This prevents a user of the object from ever modifying the value of that field.

   [Footnote: Note that a read only field is different from a constant (final) field. A read−only field can be changed by its owning object, but not by anyone else. A final field's value, once set, cannot be changed. This is enforced by the Java compiler.]

   Another example of augmenting the behavior of a setter might involve automatically filling in the city and state whenever a postal code is entered. The postal code's setter method could look up the appropriate city and state information based on the postal code supplied and propagate this information to these other fields as well, saving the user the work of providing this information separately. (Some mail order companies do this now: you give them your postal code, and they tell you what city and state you live in!)

There are other reasons why methods, rather than fields, are a good idea. Some of these involve issues that will not be discussed until later in this book. For example, if you are using inheritance (Chapter 10, *Inheritance*), methods give you additional flexibility and more appropriate behavior than fields. There are also issues that arise when two or more people try to use the same things at the same time (covered in Chapter 20, *Synchronization*); the tools that you can use to address these issues generally rely on methods rather than fields.

One of the most common reasons for a pure data repository class is to allow simultaneous return of multiple interrelated values. An example of this type is the Dimension class in the *java.awt* package. This class exists so that its instances can hold both (horizontal and

vertical) coordinates, e.g., of a window size. This allows them to be simultaneously returned from a method such as Window's *getSize* method. If *getSize* weren't able to return a data repository type such as Dimension, you'd first have to invoke a method that returned the Window's horizontal dimension, then one that returned its vertical dimension. If the Window's size changed in between these two method invocations, your two individual dimension components would combine to produce a nonsensical value!

Pure data repository objects are actually quite rare in good object−oriented design. This is because most objects do more than hold some state. The extensions we've described above, including propagation of changes, virtual fields, and access control already begin to expand the data repository idea. In the next subsection, we look at objects that exist to provide behavior without state. In the following subsection, we will return to objects that contain both data and more interesting behavior.

8.2.2 Resource Libraries

We have seen objects that hold together an interrelated set of data. Sometimes, an object exists to hold together an interrelated set of methods. If these methods are not tied to any particular state of the world, they may usefully be grouped together within a (generally non−instantiable) class that exists solely for this purpose. Consider, for example, the square root function. It is a useful function, and it is often convenient to have it lying around. But, in Java, any function must be a method belonging to a particular object. Java has a square root method; but whose method is it?

The answer to this question is that *sqrt* belongs to a special class called *Math*. ***Math*** is a class that exists precisely so that you can use its methods, like *sqrt*. Math is a canonical function library; it has no use beyond being the place to find its member functions. It exists to provide the answer to the question, “Whose method is *sqrt*?”

Because Math is a place to find these functions, it is not a class of which you would want to make instances. Instead, Math has only static methods and static fields. This means that you can use its methods and data members through the class object (Math) itself. For example, a typical method is *Math.sqrt(double d)*, which takes a `double` and returns a `double` that is the square root of its argument. Without the Math class to collect it and other mathematical functions, it is hard to imagine to whom this *sqrt* function could belong. Math exists so that there is a place to collect *sqrt* and a number of other abstract mathematical functions.

The Math class has static methods for the trigonometric functions, logarithms and exponentiation, various flavors of rounding, and very simple randomization. Math also has two (static final, i.e., constant) fields: *E* and *PI*, `doubles` representing the corresponding mathematical constants. See the sidebar on Math for details.

Question: Since it's not instantiable, why couldn't Math be an interface?

Math — the class, with its static methods and fields — is a very useful class. However, it wouldn't make sense to create any instances of it. In fact, Math has no publicly available constructor. This is a common way to prevent a class from being instantiated: give it only a private constructor. In general, a resource collection is the kind of object of which wouldn't have any use for multiple copies.

Another resource collection class is *cs101.util.Console*. Console — documented in a sidebar in the chapter on Things, Types, and Names — provides console input and output through the *print*, *println* and *readln* methods. These, too, are static methods of the class; you don't need to create a Console instance before using these methods. (In fact, like Math , Console is a class of which you can't create instances.) The resources provided by *cs101.util.Console* (streams) are a bit more complicated than the resources provided by *java.lang.Math*, and in the chapter on networking and I/O we will explore these issues in greater detail. The Console class is describe more completely in a sidebar of chapter 3.

Other classes that provide static collections of resources (whether functions or otherwise) include *java.lang.System*, *cs101.util.MoreMath*, and *cs101.util.Coerce*.

class Math

The built−in Java class Math may be the canonical resource library. It contains two (static final) fields, *Math.E* and *Math.PI*, both `doubles`, corresponding to the mathematical constants *e* and *pi*, respectively.

Math also contains a host of useful mathematical functions, again all static. Each of the following methods takes a `double` as an argument and returns a `double`:

cos	cosine of its argument
sin	sine of its argument
tan	tangent of its argument
exp	*Math.E* raised to the power of its argument
sqrt	square root of its argument
floor	largest `double` corresponding to an integer value that is smaller than its argument

acos	arc cosine of its argument
asin	arc sine of its argument
atan	arc tangent of its argument
log	Logarithm base *Math.E* of its argument
ceil	smallest `double` corresponding to an integer value that is larger than its argument
rint	the `double` corresponding to the integer value nearest its argument

Math.abs takes a `double`, a `float`, a `long`, or an `int`, and produces a value of the same type as its argument that is guaranteed to be non−negative.

Math.max and *Math.min* each take two arguments of the same type (both `double`, `float`, `long`, or `int`). The *Math.max* method returns the larger of its arguments; *Math.min* the smaller.

Math.round takes a `double` and returns the `long` closest in value to its argument.

Math.pow takes two `doubles` and yields the value of the first raised to the power of the second. That is, **`Math.pow(base, exponent)`** produces $base^{exponent}$.

Math.random takes no arguments and returns a `double` equal to or larger than 0.0 and strictly smaller than 1.0.

There are a few other Math methods not included here. In addition, there are extra mathematical functions (including more flexible and powerful randomization) available in the package *java.math*. For these additional methods, see the Java API documentation on the Javasoft web site.

8.2.3 Traditional Objects

Some objects, like data repositories, exist primarily to bundle together certain pieces of data. Other objects exist primarily to hold stateless, general−purpose functional behavior. Most objects fall into neither of these categories. Instead, most objects represent things with both state — what happens to be true of them Right Now — and behavior — how that object can change over time. Some of these objects, like Windows, Buttons, and Menus, have visual manifestations. Other objects, like the ones that represent Strings or URLs, are more obviously internal to programs. Many of the objects that you create will be of this kind.

A String is an object that keeps track of the sequence of characters of which it is composed, so somewhere inside the String object must be data that corresponds to those characters. But a String is not simply a data repository; it has a diverse set of methods. What kinds of things might you want to do with a String? Certainly look at some of the characters, which you can do using the String's *charAt* method. Java's String class provides additional methods, though, which allow you to do more than simply look at parts of the String. For example, there is *toUpperCase*, which returns a String just like the one whose method you invoke, but with all letters in upper case. (For example, **`"Hi there".toUpperCase()`** returns a String that would print out as `"HI THERE"`.) String's *toUpperCase* method is neither a selector nor a mutator. More complete descriptions of the String class and its methods are included in the sidebar on the String class in Interlude A.

Another kind of traditional object that we've seen is a counter. This object has internal state (whatever the current count is set to) and methods providing access to this state (e.g., *increment* and *getValue*. The methods can't work without the state; the state isn't directly accessible, but provides the basis for method behavior. This is an extremely typical kind of object.

Here is some code implementing a slightly more sophisticated Counter class than the one described at the beginning of this chapter. In addition to the functionality provided by that BasicCounter class, this class implements the Resettable interface, i.e., provides a *reset* method.

```
public class Counter implements Counting, Resettable {
    private int currentValue;

    public Counter() {
        this.reset();
    }

    public void increment() {
        this.currentValue = this.currentValue + 1;
    }

    public void reset() {
        this.currentValue = 0;
    }

    public int getValue() {
        return this.currentValue;
    }
}
```

The two methods — *increment* and *reset* — rely on the current state (count) of the individual instance whose methods they are. Two different counters can have two different states (e.g., one can have count 4 and the other count 27). Incrementing the first will have a different effect (producing 5, etc.) from incrementing the second (which produces 28). Resetting one will not reset the other. The *increment* and *reset* methods make no sense without reference to the particular counter instance they're incrementing or resetting. This relationship between state (data members) and methods is typical of “traditional” objects.

Picture of multiple counters

Figure 8.3. Several Counters, each with its own state.

Traditional objects are exemplified by the following properties:

- Each instance has its own state.
- This state is not directly accessible. Instead, it provides the basis for method behavior.
- Method behavior is dependent on the internal state of a particular instance.
- State plus behavior, packaged together, provide a single logical unit.

8.3 Types and Objects

8.3.1 Declared Type and Actual Type

What happens when we take an object of one type and treat it as though it had another type? One common example of this that we've seen is using an interface−type name to hold an object. The object is an instance of some class. The name says that it's in instance of some interface. The interface provides a much more limited view of the object than the actual implementation. Does this change the object? What happens when we ask whether the object is an instanceof its class, for example.

The answer is that the object is the same object no matter what its declared type (e.g. the declared type of the name that may be holding it, or of the method that may return it, or wherever else its type may be declared). It can do all of the same things regardless of its declared type. And it responds the same way when asked whether it is an instanceof its class, regardless of whether its declared type is some more specialized interface.

For example, if we take an instance of the Counter class defined above, with its *reset*, *increment* and *getValue* methods, and assign it to a name of type Counting (an interface with only *increment* and *getValue* methods), we haven't actually changed the Counter instance:

```
Counting count = new Counter();
```

If we ask whether

```
count instanceof Counter
```

this is true. Of course

```
count instanceof Counting
```

is also true. But

```
count instanceof BasicCounter
```

is false, given the definitions earlier in this chapter.

Using a Counting name instead of a Counter name does have some effect, though. First, we may not know about the Counter type. In this case, we are limited to treating *count* as though it were a Counting, not a Counter. For example, we couldn't call its *reset* method, because Countings don't have *reset* methods. Even if we did know about Counters, we'd have to explicitly cast *count* to be a Counter before we could use its Counter−specific properties:

```
((Counter) count).reset();
```

So an interface provides a limited view without limiting the actual object.

8.3.2 Use Interface Types

When declaring names and otherwise using objects, you should generally use interface types rather than class types. This allows the implementation of objects to vary independently of their use. It also allows different versions of the object to be used without dependence on unnecessary or possibly mutable properties. An interface allows common behavior to be abstracted and relied on. An interface can also be used to allow for future abstraction and variation, such as the Counting interface that allowed for the creation of a Timer.

For example, suppose that we are building a video game. The outer window of the video game is likely to be the same whether the game is Pong or Battleship or SpaceInvaders. It has controls such as *start*, *stop*, *reset*, and *pause*. What exactly happens when these controls are invoked depends on the particular game that is displayed in this window. But

we want to build a generic DefaultGameFrame window that doesn't have to rely on the particular type of game that it will hold. We can accomplish this by using an interface.

```
public interface GameControllable {
    public void start();
    public void stop();
    public void reset();
    public void pause();
    public void unpause();
}
```

Now, the DefaultGameFrame can refer to the game using the type GameControllable. As long as Pong or Battleship or SpaceInvaders implements GameControllable, any of these games can be used inside the DefaultGameFrame. When the DefaultGameFrame's reset control is invoked, DefaultGameFrame simply calls its GameControllable's *reset* method. If the GameControllable happens to be Pong, it will bring the paddles back to rest and set the scores to 0. If the GameControllable is SpaceInvaders, the player will begin again with a full set of ammunition and plenty of aliens to shoot.

8.3.3 Use Contained Objects to Implement Behavior

One object can use another to provide behavior on the first object's behalf. For example, we might have a Clock object that provides a *getTime* method and a *setTime* method. We might also have a VCR object that includes among its functionality *getTime* and *setTime*. Should the VCR implement its own *getTime* and *setTime* methods? This seems awfully inefficient. Or should the VCR reuse the Clock's *getTime* and *setTime* methods directly? (We will see a mechanism by which this can be accomplished in the chapter on Inheritance.) The problem with this solution is that the VCR isn't really a Clock (or a kind of Clock). Instead, the VCR can provide these methods by having a Clock inside it.

For example, the code for the VCR might say (in part):

```
public class VCR {

    private Clock clock;

    public Time getTime() {
        return this.clock.getTime();
    }

    public void setTime(Time t) {
        this.clock.setTime(t);
    }

    ...
}
```

In this way, the VCR provides access to the Clock's methods indirectly. This reuse of behavior by inclusion is a very powerful mechanism. In this case, the VCR might be providing access to the full set of Clock's methods. In another case, the including class might only provide a subset of the included class's methods, or it might provide a superset by combining those methods in different ways. The including class and the included class can even implement a common interface (such as TimeStorer) so that code that uses one or the other can't really tell the difference *so long as it only uses the interface's methods*.

The DefaultGameFrame and GameControllable described above are similar. When the DefaultGameFrame is asked to perform a *reset* (or a *start* or a *stop* or...), it passes this request along to the GameControllable. In that case, the use of an interface type — GameControllable — for the included object increases the flexibility and usability of the including class.

8.3.4 The Power of Interfaces

Why are interfaces so good at providing this flexibility? Because and interface is all about the contract an object makes and not about implementation. By relying on an interface, you defer any dependence on implementation details that might not be true of another implementation. This independence from implementation−specific details is enforced by the compiler, which will not let you rely on properties of an object specified by its interface type beyond those explicitly declared in the interface.

An object can also implement many different interfaces. In this case, it can be “seen” by other objects through each of these different interface types. Each interface type provides a different view of the object. By controlling these interfaces, a programmer controls the view that the object's users have of that object.

Reliance on interface types doesn't work perfectly, though. For example, a resource library such as Console or Math doesn't have an interface type. This is because resource

libraries are typically non−instantiable classes. Only instances can have interface types.

Chapter Summary

- In an informal description of the program, nouns generally correspond to objects or to fields, methods to verbs, and interfaces to adjectives.
- Classes are the factories from which objects are created.
- Interface types provide a valuable layer of abstraction, allowing the implementation to vary without affecting the use.
- Members, classes, and instance marked `public` are accessible from anywhere within a program. Members marked `private` are only accessible within their defining class or instance.
- A data repository object exists to glue together a set of interdependent data. It has fields corresponding to this data and methods that allow you to read and modify this data.
- A resource library exists to hold a collection of methods or system−wide resources. Generally, a resource library supplies these methods and resources statically, i.e., it is not a class that is ever instantiated.
- Traditional objects mix both data and methods. These objects provide the kind of integrated state−dependent behavior that we expect of real world objects.

Exercises

1. Design and implement a class called Time that keeps track of the hour and minute together. Give it a *nextMinute* method that returns another Time, a minute later. How do you access the fields of Time objects?

2. Design and implement a class that provides the following IntegerArithmetic methods:
   - *add(int, int)*
   - *sub(int, int)*
   - *mul(int, int)*
   - *div(int, int)*

   You can give it any other methods you think might be useful. What does its constructor do? Why do you think that Java doesn't have such a class?

3. Design and implement a 2DVector class representing vectors in the plane. Include *sum*, *difference*, and *product* methods.

Chapter 9

Animate Objects

Chapter Overview

- How do I create an object that can act by itself?

This chapter builds on the previous ones to create an object capable of acting without an external request. Such an object has its own instruction follower, in Java called a Thread. In addition, an object with its own instruction−follower must specify what instructions are to be followed. This is accomplished by implementing a certain interface — meeting a particular contract specification — that indicates which instructions the Thread is to execute.

The remainder of this chapter deals with examples of how Threads and animate objects can be used to create communities of autonomously interacting entities.

Objectives of this Chapter

1. To understand that Threads are Java's instruction−followers.
2. To appreciate the relationship between a Thread and the instructions that it executes.
3. To be able to construct an animate object using AnimatorThread and Animate.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

9.1 Animate Objects

In previous chapters, we saw how objects group together state and behavior. Some objects exist primarily to hold together constituent pieces of a single complex state. Other objects exist to hold a static collection of primarily functional or system−specific resources. Most objects contain both local state and methods that rely on and interact with this state in complex ways. Many of these objects wait for something to happen or for someone else to ask them to act. That is, nothing happens until something outside the object invokes a method of the object. In this chapter, we look at objects that are capable of taking action on their own, without being asked to do so from outside. These objects have their own instruction−followers, making them full−blown entities.

Consider, for example, the Counter. This is a relatively traditional object. It has both state and methods that depend on that state. An individual counter object encapsulates this state−dependent behavior, wrapping it up into a neat package. But a counter doesn't do anything unless someone asks it to, using its *increment* or *reset* method. By itself, a counter can't do much.

Contrast this with a timer. A timer is very similar to a counter in having a method that advances it to the next state (paralleling the counter's *increment* method) and one that sets the state back to its default condition (such as *reset*). A timer differs from a counter, however in that a timer counts merrily along whether someone asks it to or not. The timer's *reset* method is a traditional (passive) method; the timer resets only when asked to. But the timer's *increment* method is called by the timer itself on a regular basis.

This kind of object — one that is capable of acting without being explicitly asked to do so — is called an animate object. Such an object has its own instruction−follower, or actor, associated with it. While traditional objects are roles that an actor may take on and then leave, an animate object is a role that is almost always inhabited by an actor and tightly associated with it. Often, animate objects will use traditional objects (as well as data repositories, resource libraries, and other kinds of objects) to perform their tasks, temporarily executing instructions contained in these objects. But the animate object is where it begins and ends.

What makes an animate object different from other (passive) objects? Recall that on the first page of the first chapter of this book, we learned about the two prerequisites for a computation: The instructions for the computation must be present, and those instructions must be executed. Every method of every object is a set of instructions — a rule — that can be executed. When a method is invoked, its body is executed. (The method body is executed by the instruction−follower that invoked the method; this is how a method invocation expression is evaluated.)

An animate object differs from other objects because it also has its instruction follower. It does not need to wait for another instruction−follower to invoke one of its methods (although this may also happen). Instead, it has a way to start execution on its own.

In Java, an instruction−follower is called a Thread. No object can act except a Thread. A Thread is a special object that “breathes life” into other objects. It is the thing that causes other objects to move. An animate object is simply an object that is “born” with its own Thread. (Typically, this means that it creates its own Thread in its constructor and starts its Thread running either in its constructor or as soon as otherwise possible.)

9.2 Animacies are Execution Sequences

In every method of every object, execution of that method follows a well−defined set of rules. When the method is invoked, its formal parameters are associated with the arguments supplied to the method call. For example, recall the UpperCaser StringTransformer:

```
public class UpperCaser extends StringTransformer {
    public String transform(String what) {
        return what.toUpperCase();
    }
}
```

If we have `UpperCaser cap = new UpperCaser();` then evaluating the expression `cap.transform("Who's there?")` has the effect of associating the value of the String `"Who's there?"` with the name `what` during the execution of the body of the `transform` method.

Now, the first statement of the method body is executed. In the case of the method invocation expression `cap.transform("Who's there?")`, there is only one statement in the method body. This is the return statement, which first evaluates the expression following the return, then exits the method invocation, returning the value of that expression. To evaluate the method invocation expression `what.toUpperCase()` involves first evaluating the name expression `what` and then invoking the `toUpperCase()` method of the object associated with the name `what`.

No matter how complex the method body, its execution is accomplished by following the instructions that constitute it. Each statement has an associated execution pattern. A simple statement like an assignment expression followed by a semicolon is executed by evaluating the assignment expression. Expressions have rules of evaluation; in the case of an assignment, the right−hand side expression is evaluated, then that value is assigned to the left−hand side (dial or label). Evaluating the right−hand side expression may itself be complicated, but by following the evaluation rules for each constituent expression, the value of the right−hand side is obtained and used in the assignment.

A more complex statement, such as a conditional, has execution rules that involve the evaluation of the test expression, then execution of one *but not both* of the following substatements (the “if−block” or the “else−block”). Loops and other more complex

statements also have rules of execution. Declarations set up name−value associations; return statements exit the method currently being executed.

At any given time, execution of a particular method is at a particular point and in a particular context (i.e., with a particular set of name−value associations in force). If we could keep track of what we're in the middle of doing and what we know about while we're doing it, we could temporarily suspend and resume execution of this task at any time. Imagine that you're following an instruction booklet to assemble a complex mechanism. This problem is a lot like placing a bookmark into your instructions while you go off to do something else for a while. All you need to know is where you were, what you had around you, and what you were supposed to do next; the rest of the instructions will carry you forward.

Inside the computer, there are things that keep track of where you are in an execution sequence. These are special Java objects called Threads. The trick is that there can be more than one Thread in any program. In fact, there are exactly as many things going on at once as there are Threads executing in your program. A Thread keeps track of where it is in its own execution sequence. Each Thread works on its own assembly project using its own instruction booklet, just like multiple people can work side by side in a restaurant or a factory.

In this book, we will make extensive use of a special kind of Thread called an AnimatorThread. An AnimatorThread is an instruction follower that does the same thing over and over again. It also has some other nice properties: it can be started and stopped, suspended and resumed. These last two mean that it is possible to ask your instruction follower to take a break for a while, then ask it later to continue working. AnimatorThreads provide a nice abstraction for the kinds of activities commonly conducted by the animate objects that are often entities in our communities.

9.3 Being Animate−able

In order for a Thread to animate an object, the Thread needs to know where to begin. A Thread needs to know that it can rely on the object to have a suitable beginning place. There must be special contract between the Thread and the object whose instructions this Thread is to execute. The object promises to supply instructions; the Thread promises to execute them. (In the case of the AnimatorThread, it promises to execute these instructions over and over again.) As we know, such a contract is specified using a Java interface. This interface defines a method containing the instructions that the Thread will execute. The Thread will begin its execution at the instructions defined by this method.

9.3.1 Implementing Animate

If we use an AnimatorThread to animate our object, our object must fulfill the specific contract on which AnimatorThread begins. This contract is specified by the interface Animate:

```
public interface Animate {
    public abstract void act();
}
```

The Animate interface defines only a single method, *void act()*. A class implementing Animate will need to provide a body for its *act* method, a set of instructions for how that particular kind of object *act*'s. An AnimatorThread will call this *act* method over and over again, repeatedly asking the Animate object to *act*.

For example, the Timer that we described above could be implemented just as the Counter, but with the addition of an *act* method:

```
public void act() {
    this.increment();
}
```

Of course, we'd also have to declare that Timer implements the Animate interface. It isn't enough for Timer to have an *act* method; we also have to specify that it does so as a commitment to the Animate interface. Here is a complete Timer implementation:

```
public class Timer implements Animate {

    private int currentValue;

    public Timer() {
        this.reset();
    }

    public void increment() {
        this.currentValue = this.currentValue + 1;
    }

    public void reset() {
        this.currentValue = 0;
    }

    public int getValue() {
        return this.currentValue;
    }

    public void act() {
        this.increment();
    }
}
```

Note that the implementation is entirely identical to the implementation of Counter except for the clause `implements Animate` and Timer's *act* method.

[Footnote: As we shall see in the next chapter, we could significantly abbreviate this class by writing it as

```
public class Timer extends Counter implements Animate, Counting {

    public void act() {
        this.increment();
    }
}
```

]

Now `Timer tick = new Timer();` defines a Timer ready to be animated.

9.3.2 AnimatorThread

On the other side of this contract is the instruction follower, an AnimatorThread. Like any other kind of Java object, a new AnimatorThread is created using an instance

construction (`new`) expression and passing it the information required by AnimatorThread's constructor. The simplest form of AnimatorThread's constructor takes a single argument, an Animate whose *act* method the new AnimatorThread should call repeatedly.

For example, we can animate a Timer by passing it to AnimatorThread's constructor expression:

```
Timer tick = new Timer();
AnimatorThread mover = new AnimatorThread(tick);
```

There is one more thing that we need to do before tick starts incrementing itself: tell the AnimatorThread to *startExecution*:

```
mover.startExecution();
```

An AnimatorThread's *startExecution* is a very special method. It returns (almost) immediately. At the same time, the AnimatorThread comes to life and begins following its own instructions. That is, before the evaluation of the method invocation `mover.startExecution()`, there was only one Thread running. At the end of the evaluation of the invocation, there are two Threads running, the one that followed the instruction `mover.startExecution()` and the one named *mover*, which begins following the instructions at *tick*'s *act* method.

Once started, the AnimatorThread's job is to evaluate the expression `tick.act()` over and over again. Each time, this increments `tick`'s `currentValue` field. The AnimatorThread named `mover` calls `tick`'s *act* method over and over again, repeatedly causing `tick` to `act`.

We can collapse the two AnimatorThread statements into one by writing

```
new AnimatorThread(tick).startExecution();
```

However, this form does not leave us holding onto the AnimatorThread, so we couldn't later tell it to *suspendExecution*, *resumeExecution*, or *stopExecution*. (See below.) If we anticipate needing to do any of these things, we should be sure to hold on to the AnimatorThread (using a label name).

9.3.3 Creating the AnimatorThread in the Constructor

If our Timers will always start ticking away as soon as they are created, we can include the Thread creation in the Timer constructor:

```
public class AnimateTimer implements Animate {

    private int currentValue;
    private AnimatorThread mover;

    public AnimateTimer() {
        this.reset();
        this.mover = new AnimatorThread(this);
        this.mover.startExecution();
    }

    public void increment() {

    // ... rest of class is same as Timer
```

In this case, as soon as we say

```
Timer tock = new AnimateTimer();
```

`tock` will begin counting away. If we invoke `tock.getValue()` at two different times — even if no one (except its own AnimatorThread) asks `tock` to do anything at all in the intervening time — the second value might not match the first. This is because `tock` (with its AnimatorThread) can act without needing anyone else to ask it.

Here is another class that could be used to monitor a Counting (such as a Counter or a Timer):

```
public class CountingMonitor implements Animate {

    private Counting whoToMonitor;
    private AnimatorThread mover;

    public CountingMonitor(Counting whoToMonitor) {
        this.whoToMonitor = whoToMonitor;
        this.mover = new AnimatorThread(this);
        this.mover.startExecution();
    }

    public void act() {
        Console.println("The timer says "
                        + this.whoToMonitor.getValue());
    }
}
```

Note in the constructor that the first whoToMonitor (this.whoToMonitor) refers to the field, while the second refers to the parameter.

9.3.4 A Generic Animate Object

The way that AnimateTimer and CountingMonitor use an AnimatorThread is pretty useful. There is a cs101 class, AnimateObject, that embodies this behavior. It is probably the most generic kind of animate object that you can have; any other animate object would behave like a special case of this one. We present it here to reinforce the idea of an independent animate object. It generalizes both CountingMonitor and AnimateTimer.

At this point, you should regard this class as a template. Change its name and add a real *act* method to get a real self−animating object. In the chapter on Inheritance, we will return to this class and see that there is a way to make this template quite useful directly.

```
public class AnimateObject implements Animate {

    private AnimatorThread mover;

    public AnimateObject() {
        this.mover = new AnimatorThread(this);
        this.mover.startExecution();
    }

    public void act() {
        // what the Animate Object should do repeatedly
    }
}
```

It is worth noting that an Animate need not be animated by an AnimatorThread. For example, a group of Animates could all be animated by a single SequentialAnimator that asks each Animate to *act*, one at a time, in turn. No Animate could *act* while any other Animate was mid−*act*. Each would have to wait for the previous Animate to finish. This SequentialAnimator would require only a single instruction follower (or Thread) to execute the sequential Animates' instructions, because it would execute them one *act* method at a time. When one animate is acting, no one else can be.

The nature of execution under such a synchronous assumption would be very different from executions in which each Animate had its own Thread and they were all acting simultaneously. Roughly it's the difference between a puppet show with one not−very−skillful puppeteer, who can only operate a single puppet at a time, and a whole crowd of puppeteers each operating a puppet. The potential for chaos is much greater in the second scenario, but so is the potential for exciting interaction. When each object has its own AnimatorThread — as in the AnimateObject template — any other Animate (or the methods it calls) can execute at the same time.

9.4 More Details

This section broadens the picture painted so far.

9.4.1 AnimatorThread Details

The AnimatorThread class and the Animate interface reside in the package cs101.lang. This means that any file that uses these classes should have the line

```
import cs101.lang.*;
```

before any class or interface definition.

The class AnimatorThread specifies behavior for a particular kind of instruction follower. Its constructor requires an object that implements the interface cs101.lang.Animate, the object whose *act* method the AnimatorThread will repeatedly execute.

After constructing an AnimatorThread, you need to invoke its *startExecution* method.

[Footnote: AnimatorThread's instances also have a *startExecution* method that is identical to the *startExecution* method. This is for historical reasons.]

This causes the AnimatorThread to begin following instructions. In particular, the instructions that it follows say to invoke its Animate's *act* method, then wait a little while, then invoke the Animate's *act* method again (and so on). To temporarily suspend execution, use the AnimatorThread's *suspendExecution* method. Execution may be restarted using *resumeExecution*. To permanently terminate execution, AnimatorThread has a *stopExecution* method. Once stopped, an AnimatorThread's execution cannot be restarted. However, a new AnimatorThread can be created on the same Animate object.

An object — like an Animate — is a set of instructions — or methods — plus some state used by these instructions. There is nothing to prevent more than one Thread from following the same set of instructions at the same time. For example, it would be possible to start up two AnimatorThreads on the same Timer. If the two AnimatorThreads took turns fairly and evenly, one AnimatorThread would always move from an odd to an even numbered currentValue, while the other would always move from an even to an odd numbered value. Of course, there's nothing requiring that the two AnimatorThreads play fair. Like children, one might take all of the turns — incrementing the Timer again and again — while the other might never (or rarely) get a turn. AnimatorThreads are designed to minimize this case, but it can happen. The problem is more prevalent with other kinds of Threads.

One of the ways in which AnimatorThread tries to "play fair" is in providing intervals between each attempt to follow the *act* instructions of its Animate object. The AnimatorThread has two values that it uses to determine the minimum interval between

invocations of the Animate's *act* method and the maximum interval. Between these two values, the actual interval is selected at random each time the AnimatorThread completes an *act*. You can adjust these parameters using setter methods of the AnimatorThread. Values for these intervals may also be supplied in the AnimatorThread's constructor. See the AnimatorThread sidebar for details.

class AnimatorThread

AnimatorThread is a cs101 class (specifically, cs101.lang.AnimatorThread) that serves as a special kind of instruction−follower. An AnimatorThread's constructor must be called with an instance of cs101.lang.Animate. The AnimatorThread repeatedly follows the instructions in the Animate's *act* method.

An AnimatorThread is an object, so it can be referred to with an appropriate (label) name. It also provides several useful methods:

> `void startExecution()` causes the AnimatorThread to begin following the instructions at its Animate's *act* method. Once started, the AnimatorThread will follow these instructions repeatedly at semi−random intervals until it is stopped or suspended
>
> `void stopExecution()` causes the AnimatorThread to terminate its execution. Once stopped, an AnimatorThread cannot be restarted. This method may terminate execution abruptly, even in the middle of the Animate's *act* method.
>
> `void suspendExecution()` causes the AnimatorThread to temporarily suspend its execution. If the AnimatorThread is already suspended or stopped, nothing happens. If the AnimatorThread has not yet started and is started before an invocation of *resumeExecution*, it will start in a suspended state, i.e., it will not immediately begin execution. This method will not interrupt an execution of the Animate's *act* method; suspensions take effect only between *act*'s.
>
> `void resumeExecution()` causes the AnimatorThread, if suspended, to continue its repeated execution of its Animate's *act* method. If the AnimatorThread is not suspended or already stopped, this method does nothing. If the AnimatorThread is suspended but not yet started, invoking *resumeExecution* undoes the effect of any previous *suspendExecution* but does not *startExecution*.

Between calls to the Animate's *act* method, the AnimatorThread sleeps, i.e., remains inactive. The duration of each of these sleep intervals is randomly chosen to be at least `sleepMinInterval` and no more than `sleepMinInterval + sleepRange`. These values are by default set to a range that allows for variability and slows activity to a rate that is humanly perceptible. If you wish to change these defaults, they may be set either explicitly using setter methods or in the AnimatorThread constructor.

> `void setSleepRange(long howLong)` sets the desired variance in sleep times above and beyond sleepMinInterval
>
> `void setSleepMinInterval(long howLong)` sets the range of variation in the randomization

By setting sleepRange to 0, you can make your AnimatorThread's activity somewhat more predictable as it will sleep for approximately the same amount of time between each execution of the Animate's *act* method. Setting sleepMinInterval to a smaller value speeds up the execution rate of the AnimatorThread. Setting it to 0 can be dangerous and should be avoided. If sleepRange is 0, it is possible that this AnimatorThread will interfere with other Threads' ability to run.

AnimatorThread supplies a number of constructors. The first requires only the Animate whose act method supplies this AnimatorThread's instructions:

```
AnimatorThread(Animate who)
```

The next two constructors incorporate the same functions as setRange and setMinInterval:

```
AnimatorThread(Animate who, long sleepRange)

AnimatorThread(Animate who, long sleepRange,
                            long sleepMinInterval)
```

It is also possible to specify explicitly whether the AnimatorThread should start executing immediately. By default, it does so. The following constructor allows you to override this explicitly using the boolean constants `AnimatorThread.START_IMMEDIATELY` and `AnimatorThread.DONT_START_YET`.

```
AnimatorThread(Animate who, boolean startImmediately)
```

Finally, there are two additional constructors that incorporate both startup and timing information:

```
AnimatorThread(Animate who, boolean startImmediately,
                            long sleepRange)

AnimatorThread(Animate who, boolean startImmediately,
                            long sleepRange,
                            long sleepMinInterval)
```

9.4.2 Delayed Start and the *init* Trick

It is awfully convenient to be able to define an animate object as an Animate that creates and starts its own AnimatorThread. This hides the Thread creation and manipulation inside the Animate (as in the example of AnimateTimer), making it appear to be a fully self−animating object from the outside. However, sometimes we need to separate the construction of the Animate and its AnimatorThread from the initiation of the AnimatorThread instruction follower. That is, we want the AnimatorThread set up, but not yet actually running. For example, we might need a part that isn't yet available at Animate/AnimatorThread creation time. On these occasions, it would be awkward to start the execution of an AnimatorThread in the constructor of its Animate. For example, if the Animate's *act* method relies on other objects and these other objects may not yet be available, you wouldn't want the AnimatorThread to start executing the *act* method yet.

An example of this might be in the StringTransformer class in the first interlude, in which you can't read or transform a String until after you've accepted an input connection. Since the input connection might not be available at StringTransformer construction time, one solution to this problem is to delay the starting of the execution of the *act* method until after the input connection has been accepted. Once the constructor completes, the newly constructed object's acceptInputConnection method can be invoked. At this point — and not before — the AnimatorThread's *startExecution* method can be invoked. This means that the call to the AnimatorThread's *startExecution* method can't appear in the constructor. But it can't be invoked by any object other than the Animate, because the AnimatorThread is held by a private field of the Animate.

This situation — that there are things that need to be done that are logically part of the setup of the object, but that cannot be done in the constructor itself — is a common one. To get around it, there is a convention that says that such objects should have *init* methods. Whoever is responsible for setting up the object should invoke its *init* method after this setup is complete. The object can rely on the fact that its *init* method will be invoked after the object is completely constructed and — in this case — connected. We could then put the call to the AnimatorThread's *startExecution* method inside this *init* method.

Here is a delayed−start version of the AnimateObject template.

```
public class InitAnimateObject implements Animate {

    private AnimatorThread mover;

    public InitAnimateObject() {
        this.mover = new AnimatorThread(this);
    }

    public void init() {
        this.mover.startExecution();
    }

    public void act() {
        // what the Animate Object should do repeatedly
    }
}
```

A concrete example of this issue arises if we look at CountingMonitor and don't assume that the Counting will be supplied to the constructor. Here is another version of CountingMonitor without the constructor parameter:

```
public class InitCountingMonitor implements Animate {

    private Counting whoToMonitor;
    private AnimatorThread mover = new AnimatorThread(this);

    public void setCounting(Counting whoToMonitor) {
        this.whoToMonitor = whoToMonitor;
    }

    public void init() {
        this.mover.startExecution();
    }

    public void act() {
        Console.println("The timer says "
                        + this.whoToMonitor.getValue());
    }
}
```

The use of a method named *init* here is completely arbitrary. You are free to define your own method and call it whatever you want. However, you will see that many people follow this convention and provide an *init* method for their objects when there is initialization that must take place after the constructor and setup process is complete.

9.4.3 Threads and Runnables

The Animate/AnimatorThread story that we've just seen is not a standard part of Java, though it is only a minor variant on something that is. There are two reasons why we've used AnimatorThreads here. The first is that most of the self−animating object types in this book are objects whose act method is executed over and over again. AnimatorThread is a special kind of Thread designed to do just that. The second is that AnimatorThread contains some special mechanisms to facilitate its use in applications where you might want to suspend and resume its execution or even to stop it entirely. AnimatorThread provides methods supporting this behavior.

There is, however, in Java a more primitive type of Thread, called simply Thread. Like an AnimatorThread, a simple Java Thread can be given an object to animate when the Thread is created. (Its constructor takes an argument representing the object whose instructions the Thread is to follow once it has been started.) However, the Thread does not execute this method repeatedly; it executes it once, then stops. The contract that a Thread requires of the object providing its instructions is not Animate, meaning it can be called on to act repeatedly. Instead, it is Runnable, meaning it can be executed once.

Thread (as of Java 1.1) does not provide suspension, resumption, or cessation methods. In this book, we avoid the use of plain Java Threads.

In addition, it is technically possible in Java to extend a Thread object rather than passing it an independent Runnable. Except in code that creates special kinds of Threads (such as AnimatorThread) capable of animating other objects, the extending of Thread is highly discouraged in this book. Extending Thread to create an executing object (whose own *run* method is the set of instructions to be followed) confounds the notion of an executor with the executed.

9.4.4 Thread Methods

Thread methods

Threads are Java's instruction followers. In this book, we will most often make use of AnimatorThread's. However, it is useful to understand how Java's built−in Thread class works as well.

Like an AnimatorThread, each Thread provides a few methods for its management.

> **`void start()`** Like AnimatorThread's *startExecution*, this method causes the target Thread to begin following instructions. If the Thread's constructor was supplied a Runnable, the Thread begins execution at this Runnable's *run* method. When the *run* method

terminates, the Thread's execution is finished.

`boolean isAlive()` tells you whether the target Thread is alive, i.e., has been started and has not completed its execution.

`void interrupt()` sends the target Thread an InterruptedException. Useful if that Thread is sleeping, waiting, or joining.

`void join()` causes the invoking Thread to suspend its execution until the target Thread completes. Variants allow time limits on this suspension: `void join(long millis)` and `void join(long millis, long nanos)`.

Unlike AnimatorThread, a Thread cannot safely be stopped, suspended, or resumed.

In addition to its role as the type of Java's instruction followers, the Thread class provides useful static (i.e., class−wide) functionality. These methods are static methods of the class Thread:

`static void sleep(long millis)` causes the currently active Thread to stop executing for millis milliseconds. This method throws InterruptedException, so it cannot be used without some additional machinery (introduced in the chapter on Exceptions). There is a variant method, `sleep(long millis, long nanos)` that allows more precision in controlling the duration of the Thread's sleep.

`static void yield()` is intended to pause the currently executing Thread and to allow another Thread to run. However, not all versions of Java implement Thread.yield in a way that ensures this behavior.

Other Thread features are outside the scope of this course.

9.5 Where Do Threads Come From?

We have discussed the idea of AnimatorThreads above, showing how to create self−animating objects by having an AnimatorThread created in an object's constructor. Such an object is born running; it continually acts, over and over, until its Thread is suspended or stopped.

In fact, no execution in Java can take place without a Thread. But something must call the AnimatorThread constructor; this instruction must be executed by a Thread! So where does the first Thread come from? This depends on the particular kind of Java program that you are running. In this book, we look primarily at Java applications. In the appendix, we also answer these questions for Java applets.

9.5.1 Starting a Program

What does it mean for a Java program to run? It means that there is an instruction follower that executes the instructions that make up this program. In Java, there is no execution without a Thread, or instruction−follower, to execute it. So when a program is run, some Thread must be executing its instructions. Where does this Thread come from, and how does it know what instructions to execute?

Let's answer the first of these questions first. When a Java program is run, a single Thread is created and started. This is not a Thread that your program creates; it is the Thread that Java creates to run your program. Depending on whether your Java program is an application (as we're discussing in this book) or an applet (as you may have encountered on the world−wide web) or some other kind of Java program, there are different conventions as to where this Thread begins its execution. But running a program *by definition* means creating a Thread — an instruction follower — to execute that program.

How does the Thread know where to begin? By convention. What do we mean by a convention? AnimatorThread's use of Animate is a convention. This convention is, in some sense, completely arbitrary. That is, a different interface name or other method might have been used. For example, the raw Thread class uses a different convention, that of Runnable/*run*. If you were to design your own type of Thread, you could create a different convention for it to follow. However, once these names and contracts have been selected by the designers of AnimatorThread and Thread, they are absolute rules that cannot be violated.

Similarly, there must be some arbitrary convention as to how a Java program begins. In a standalone application, the convention is that running a Java program means supplying a class to the executable, and by convention a particular method of the class is always the place that execution begins. This default execution does not create an instance of the class, so the method must be a static one. Again by convention, the name of this method is *main*, it takes as argument an array of Strings, and it returns nothing. That is, the arbitrary but unvarying start point for the execution of a standalone Java application is the

```
public static void main(String[] args)
```

method of the class whose name is supplied to the executable.

[Footnote: Typically, this means the class you select before choosing run from the IDE

menu or the class whose name follows the command `java` on the command line.]

So if you want to write a program, you simply need to create a class with a method whose signature matches the line above. The body of that *main* method will be executed by the single Thread that is created at the beginning of a Java execution. When execution of *main* terminates, the program ends. If you do not want the program to end, you need to do something during the course of executing *main* that causes things to keep going. Typically, this means that you use the body of *main* to create one or more objects that themselves may execute. For example, if the body of *main* creates an animate object (with its own AnimatorThread), then that object will continue executing even if the body of *main is completed. This is called "spawning a new Thread."*

Here is a very simple class that exists solely to create a new instance of the AnimateTimer class:

```
public class Main {

    public static void main(String[] args) {
        Counting theTimer = new AnimateTimer();
    }
}
```

This program simply counts. The instruction follower that begins when this program starts up (e.g., using the command **`java Main`**) executes the *main* method, invoking **`new AnimateTimer()`** and assigning the result to *theTimer*. This Thread is now done executing and stops. However, the constructor for AnimateTimer has created a new AnimatorThread and then called that AnimatorThread's *startExecution* method. This starts up the new Thread which repeatedly calls AnimateTimer's *act* method. The program as a whole will not terminate until the AnimatorThread stops executing, which it will not do by itself. If you run this program, you will need to forcibly terminate it from outside the program!

Since we didn't give this program any way to monitor or indicate what's going on, running it wouldn't be very interesting. But we can use the CountingMonitor above to improve this program:

```
public class Main {

    public static void main(String[] args) {
        Counting theTimer = new AnimateTimer();
        Animate theMonitor = CountingMonitor(theTimer);
    }
}
```

Question: Can you find a more succinct way to express the body of the *main* method?

Question: What will be printed by this program? On what does it depend? (Hint: fairness.)

The instruction follower executing the Main class's *main* method exits. However, before it completes it executes the instructions to create and start two separate AnimatorThreads. These AnimatorThreads continue after the execution of the *main* Thread exits. Again, this program must be forcibly terminated from outside.

Question: Can you cause this program to stop by itself sometime after it has counted to 100? (This is a bit tricky.)

The two versions of the Main class above each contain just the instructions to create an instance or two. In the cs101 libraries, we have provided a Main that does this for you. This allows you to write applications without needing to write **`public static void main(String[])`** methods yourself.

class Main

The cs101 libraries include a class, cs101.util.Main, that can be run from the java command line to create an instance of a single class with a no−args constructor. For example, we could implement the unmonitored Timer example using the following command:

```
java  cs101.util.Main  AnimateTimer
```

This causes code much like the first Main class to execute, creating a single instance of AnimateTimer (using its no−args constructor).

The class cs101.util.Main contains nothing but the single static method *main* (taking a String[] argument). The command above tells Java to start its initial instruction follower at this method — the static *main(String[])* method of the class cs101.util.Main. The remainder of the information on the command line (in this case, AnimateTester) is supplied to the *main* method using its parameter.

[Footnote: For more detail on arrays ([]), see the chapter on Dispatch.]

Style Sidebar

Using *main*

If you do decide to write your own *main* method, you should do so in a class separate from your other classes, generally one called Main and containing only the single public static void *main* method requiring a String[] (i.e., an array of Strings). This method may have some complexity, creating several objects and gluing them together, for example.

Alternately, you can create an extremely simple *main* method in any (or even every) class that you write. In this case, however, the *main* method should do nothing more than to create a single instance of the class within which it is defined, using that class's no−args constructor. Of course, the signature of each *main* method is the same:

```
public static void main(String[] args)
```

The *main* that will actually be executed is the one belonging to the (first) class whose name is supplied to the java execution command. So, for example, in the sidebar on class Main, we said

```
java  cs101.util.Main  AnimateTimer
```

causing cs101.util.Main's *main* method to be run.

The logic behind these restrictions on the use of *main* is as follows. In the second case — *main* in many instantiable class's files — the presence of *main* allows that object to be tested independently. However, this test is extremely straightforward and predictable. If the *main* method takes on any additional complexity, it should be separated from the other (instantiable) classes and form its own resource library, one that exists solely to run the program in all its complexity.

9.5.2 Why Constructors Need to Return

In the code above, each Animate's constructor calls the *startExecution* method of a new Thread. This in turn repeatedly calls the *act* method of the Animate. Why doesn't the constructor just repeatedly call the Animate's *act* method itself (e.g., in a `while` loop)?

This is a fundamental issue. If the Animate's constructor called the *act* method itself, the instruction follower — or Thread — executing the constructor would be trapped forever in a loop calling *act* over and over. The constructor invocation — the new expression —

would never complete. In the monitored counting example, the invocation of AnimateTimer's constructor would cause the instruction follower to execute the *act* method of AnimateTimer over and over again. This instruction follower — the only instruction follower to be running so far — would never complete the repeated execution of the *act* method. This means that it would never get around to creating the CountingMonitor.

This is why *AnimatorThread.startExecution* must be a very special kind of method. The Thread, or instruction follower, that executes *startExecution* must return (almost) immediately. It is the *new* Thread, the one just started, that goes off to execute the *act* method. The original Thread returns from this invocation and goes about its business just as if nothing ever happened. In personal terms, this is the difference between doing the job yourself and assigning someone else to do it. True, when someone else does it you have less control over how or when the job gets done; but while someone else is working on it, you can be doing something else.

Chapter Summary

- In Java, activity is performed by instruction followers called Threads.
- An animate object is simply one that has its very own Thread.
- An AnimatorThread is a useful kind of Thread that repeatedly follows the instructions provided by some object's *act* method.
    - ♦ This object must implement the Animate interface.
    - ♦ It must be supplied to the AnimatorThread's constructor.
- An AnimatorThread can also be asked to start, stop, suspend, or resume execution.
- Java programs may involve other Threads.
    - ♦ One Thread begins execution at

      ```
      public static void main(String[] args)
      ```

      when a Java application is begun.
    - ♦ GUI objects involve their own Threads.
    - ♦ Other Threads may be explicitly created.

Exercises

1. Define a class whose instances each have an internal value that doubles periodically. Each time that the value doubles, the instance should print this new value to the Console.

2. Define a class that periodically reads from the Console and writes the value back to the Console.

3. Define a *main* class that creates three instances of your doubler.

4. Using the timing parameters of AnimatorThread, demonstrate that not all doublers have to run at the same rate.

Chapter 10

Inheritance

Chapter Overview

- How do I simplify the program design task by reusing existing code?
- How do I create variants on things I already have?
- When is it not appropriate to reuse code?

This chapter covers class−based ***inheritance*** as a way to reuse implementation. Inheritance allows you to define a new class by specifying only the ways in which it differs from an existing class. Those differences can include: additional (or alternative) contracts that it satisfies, behaviors that it provides, internal information that it stores, or startup instructions. Inheritance means that existing code can be adapted and reused, with some modification, in new contexts.

The mechanism by which inheritance works involves extending the parent class definition either by augmenting or overriding behavior defined there. Most of this chapter concentrates on how these mechanisms work. Not every instance of similar behavior is an appropriate context for inheritance. The chapter concludes with a discussion of the limitations of inheritance.

This chapter includes sidebars on the details of method and field lookup. It is supplemented by reference charts on the syntax and semantics of java methods, fields, and class declarations.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Objectives of this Chapter

1. To understand how one class can build on behavior supplied by another.
2. To be able to extend and modify existing definitions
3. To recognize when to use mechanisms other than inheritance to extend behavior.

10.1 Derived Factories

We have so far seen several cases in which we wanted to build multiple kinds of things that shared a basic similarity. When this similarity was largely in the contract implemented — as with Counters and Timers — we abstracted this similarity into an interface. The interface allowed us to deal with objects without knowing the details of their implementations, i.e., to treat them solely in light of the contracts that they provided.

In this chapter, we are more concerned with situations in which two kinds of objects share not only the same contract but almost the same implementation. For example, the BasicCounter and the Resettable Counter contained almost precisely the same code. In fact, the BasicCounter's code was (except for the class and constructor name) a proper subset of the Resettable Counter's code. Similarly, the code for AnimateObject was contained in the code for AnimateTimer and the code for CountingMonitor. And almost every StringTransformer simply elaborates on the generic StringTransformer, simply providing a specialized version of the *transform* method.

In cases where code really matches at the level of wholesale textual reuse of a class, Java provides a mechanism to allow one type of object to build on the behavior specified by another. This is a relationship between one class and another. Since classes are essentially object factories, we can think of this as a situation in which one factory produces its widgets by buying widgets wholesale from another factory, then adding its own minor tweaks (bells and whistles) to the widgets before claiming to have produced them.

The mechanism by which this is accomplished in Java is called ***inheritance***, and it applies to a relationship between two classes. There is a similar relationship between two interfaces, described below. Inheritance is *not* ever a relationship between a class and an interface (or between an interface and a class). Inheritance really means an almost literal subsuming of one thing by another.

10.1.1 Simple Inheritance

Consider, for example, the ***AnimateObject*** class from the previous chapter and its near relative, the ***CountingMonitor***. The AnimateObject class says:

```
public class AnimateObject implements Animate {

    private AnimatorThread mover;

    public AnimateObject() {
        this.mover = new AnimatorThread(this);
        this.mover.startExecution();
    }

    public void act() {
        // what the AnimateObject should do repeatedly
    }
}
```

In implementing the CountingMonitor class, we really only want to change the highlighted things:

```
public class CountingMonitor implements Animate {

    private Counting whoToMonitor;

    private AnimatorThread mover;

    public CountingMonitor(Counting whoToMonitor) {
        this.whoToMonitor = whoToMonitor;
        this.mover = new AnimatorThread(this);
        this.mover.startExecution();
    }

    public void act() {
        Console.println("The timer says "
                          + this.whoToMonitor.getValue());
    }
}
```

It would be really nice only to have to write the highlighted information, not the rest. In fact, we can do almost exactly that. The following definition is *almost* equivalent to the Counter definition above:

```
public class CountingMonitor extends AnimateObject {

    private Counting whoToMonitor;

    public CountingMonitor(Counting whoToMonitor) {
        this.whoToMonitor = whoToMonitor;
    }

    public void act() {
        Console.println("The timer says "
                         + this.whoToMonitor.getValue());
    }
}
```

We have preserved the highlighting, and you can see that almost the entire new class is highlighted. One of the few non−highlighted items is the phrase **`extends AnimateObject`**. This is the phrase that does almost all of the work. It means, roughly, a CountingMonitor *is* an AnimateObject; it just provides the additional specified behavior.

1. It has its own private field, *whoToMonitor*, suitable for labeling a Counting.

2. It has a constructor that takes one argument, a Counting, and holds on to it.

3. Its *act* method has a much more interesting body than AnimateObject's.

This code is equivalent to the original definition of CountingMonitor. It is much shorter to write. To use it, simply begin with the instructions for AnimateObject and add the pieces that CountingMonitor provides, ***extending*** the behavior of the AnimateObject (in the absence of conflicting instructions) to do these additional things.

In essence each CountingMonitor instance has an AnimateObject instance inside of it. Whenever the CountingMonitor can't figure out how to do something, it simply defaults to the behavior of its AnimateObject. That way, the CountingMonitor doesn't have to provide all of the behavior that an AnimateObject already has; it can just rely on the existing implementation.

The remainder of this chapter deals with the details of this proposition.

10.1.2 The *java.lang.Object* Type

There is actually a single built−in type called ***Object***, and all other object types (directly or indirectly) ***extend Object***. In other words, anything which is not one of the built in types is an Object of some sort or another.

```
class Cat extends Animal {
    ....
}
```

A class declaration is followed by an optional `extends` clause, then a pair of braces around the body of the class definition. If the `extends` clause is missing (e.g., **`class Widget {...}`**), the default clause **`extends Object`** is assumed. Thus, **every class** (implicitly or explicitly, directly or indirectly) **extends Object**.

The class Object provides some basic functionality that every other class necessarily inherits. This means that you can guarantee that every Java object has, e.g., a *toString* method. See the sidebar on *The Object class* for details.

The Object class

The class *java.lang.Object* is the root of the inheritance hierarchy, i.e., the class of which all other classes are subclasses. Every Java object is guaranteed to implement each of the methods provided by Object (though their implementations may vary).

> ***equals(Object)*** returns `true` exactly when the argument Object is the same Object as the one whose method is invoked. This is exactly the same thing that == would do on two Objects. You may override *equals* to do something somewhat more interesting.
>
> ***toString()*** returns a String ostensibly suitable for printing. It contains a lot of useful information in a generally illegible format, so if you are interested in being able to read your objects, you may wish to override this method to print something more easily human−readable.
>
> ***getClass()*** returns the Class object (i.e., factory) from which this instance was created.
>
> ***clone()*** is a peculiar method of Object because although every object implements it, it can be used only with instances of classes that also implement the ***Cloneable*** interface. If a class implements the Cloneable interface, the inherited version of *clone* simply creates a new object of the same type as the original and whose fields have the same values as the fields of the original. You may override *clone* to do whatever you wish.
>
> [Footnote: If you call the *clone* method of an object that doesn't implement Cloneable, it will throw ***CloneNotSupportedException***. See the next chapter for more on Exceptions.]

Object also provides other methods (*finalize*, *hashcode*, *wait*, *notify*, and *notifyAll*) that are beyond the scope of the material covered here.

10.1.3 Superclass Membership

When one class extends another — as in the CountingMonitor / AnimateObject example above, we say that the extending class (CountingMonitor) is a ***subclass*** of the extended class (AnimateObject), and that the extended class is a ***superclass*** of the extending class. Neither subclass nor superclass is an absolute description; instead, both describe relationships between two classes.

When we say that one class is a subclass of another, what we mean is that we can treat instances of the subclass in all respects as though they were members of the superclass. For example, we can use a CountingMonitor anywhere we can use an AnimateObject. We can assign a CountingMonitor to a name whose type makes it appropriate for labeling AnimateObjects. (After all, a CountingMonitor *is* an AnimateObject.) We can return a CountingMonitor from a method that expects to return an AnimateObject, or pass one as an argument to a method expecting an AnimateObject parameter. A CountingMonitor is simply a special kind of AnimateObject.

In fact, subclasses have all of the type−relational properties of classes and the interfaces that they implement. A subclass instance can be assigned to a name of the superclass type. It answers `true` to the *instanceof* predicate on the superclass.

[Footnote: A ***predicate*** is a boolean method, that is, a method whose returned value is `true` or `false`.]

It can even be automatically coerced ***up−cast*** to its superclass type. This is the same kind of automatic coercion that happens from int to long, and it is similarly guaranteed always to succeed and never to lose information.

Treating a CountingMonitor as an AnimateObject doesn't actually change the CountingMonitor, though. The CountingMonitor is still a CountingMonitor, with its extended *act* method and its Counting to keep track of. This is the same situation as when an object is treated according to its interface type: this narrows the view of the object, but it doesn't change the underlying object.

If you are currently holding what looks like a superclass instance (e.g., an AnimateObject), and you suspect that it is actually an instance of a subclass, you can attempt to do a ***down−cast*** coercion on it. As with primitive types, a narrowing conversion is one that may not work or may lose information.

For example, if AnimateObject *ao* has some value that you think might be a CountingMonitor, you can try the expression

```
(CountingMonitor) ao
```

(e.g., in an assignment statement or in a method invocation). However, if you're wrong and this AnimateObject is not a CountingMonitor, this will cause your program serious problems. (See the next chapter for information about how these problems arise and what you can do about them.) So you may want to test whether this is an OK thing to do first, using a ***guard expression***:

```
CountingMonitor cm;
if (ao instanceof CountingMonitor) {
    cm = (CountingMonitor) ao;
}
```

This first checks to see whether it's OK to treat the AnimateObject as a CountingMonitor.

So far, we have seen that instances have several types: the type of the class from which the instance was created, the types of any interfaces that class implemented, and the types of any superclass that this class extends. This may mean many interface types (since a class can implement many interfaces). A class can only extend a single superclass, but this does not limit the number of legal class types because the superclass may itself extend another class, and so on. Where does this end?

We can use the idea of superclass membership to create very powerful abstractions, but not without the help of casting. For example, Java provides a class, ***Vector***, that allows us to hold on to a collection of Objects; it behaves sort of like a whole bunch of names, but indexed by number. Vector provides an *addElement* method that takes any Object as an argument. This means that any Object can be inserted into a Vector. For example, you can insert a String into a Vector, and an AnimateObject as well:

```
Vector v = new Vector();
v.addElement("Silly string");
v.addElement(new Timer());
```

However, when we retrieve the elements we've inserted, we discover that Vector's *elementAt* method doesn't know the type of the Object we've inserted. Instead, elementAt() returns an Object; it is up to us to figure out what kind of thing we've gotten back. For example, the first thing in the Vector (at element 0) is the String `"Silly string"`. So we can say

```
Object o = v.elementAt(0);
```

or

```
String s = (String) v.elementAt(0);
```

but not

```
String s = v.elementAt(0);
```

because this is an illegal attempt to assign a value of type Object (v.elementAt(0)) to a name of type String. The explicit cast expression of the previous line is needed to make this statement legal.

10.2 Overriding

The examples of inheritance in the previous section demonstrated that a subclass can extend the functionality of its superclass. The subclass can also modify superclass functionality by ***overriding***, or redefining, methods provided by the superclass. In fact, CountingMonitor overrode the *act* method provided by AnimateObject. This just wasn't a

very interesting example because AnimateObject's *act* method didn't do anything.

Consider the following classes:

```
public class Super {
    public void doit() {
        Console.println("super method");
    }

    public void doitAgain() {
        this.doit();
    }
}

public class OverridingSub extends Super {
    public void doit() {
        Console.println("overridingSub method");
    }
}
```

Now suppose that we create an instance of OverridingSub and ask it to doit():

```
OverridingSub over = new OverridingSub();
over.doit();
```

As expected, this prints `overridingSub method`. What if we labeled the OverridingSub with a Super name?

```
Super supe = new OverridingSub();
supe.doit();
```

The same thing: `overridingSub method`. Recall that using a different type of name doesn't change the underlying object.

10.2.1 The *super* Expression

What if we still want to be able to access Super's *doit* method from the subclass? To do this, we need a special expression much like `this`. The expression `this` refers to the instance whose code is being executed. The expression ***super*** refers to the superclass of the object containing the actual executing code.

```
public class ExpandingSub extends Super {
    public void doit() {
        super.doit();
        Console.println("expandingSub method");
    }
}
```

In this case, we'll get the effect of executing the superclass method followed by the local *println*:

```
super method
expandingSub method
```

If we reverse the lines of the method body, we will reverse the order of the printed lines.

10.2.2 The Outside−In Rule

There is one more trick lurking in this example. This is the *doitAgain* method in Super. We know what happens when we ask an instance of Super to **`doitAgain()`**: it does the same thing as if we'd asked it to **`doit()`**. But what if we ask a subclass instance?

```
over.doitAgain()
```

The first thing that happens is that we have to find the *doitAgain* method for OverridingSub. To do this, we start looking at the outermost (sub) class. This is OverridingSub. But it doesn't contain an appropriate method. So we move up the hierarchy, inside the object, to the superclass. Super *does* define *doitAgain*, so now we know what code to execute. But the body of Super's *doitAgain* method says **`this.doit()`**. Who is *this*?

The expression ***this*** always refers to the object on behalf of whom you are executing. At the moment, we're executing some code in the class Super. But we are doing it for an instance of OverridingSub; we just happen to be looking at *over as though* it were a Super, just as we did when we labeled it with a Super−type name. Looking at *over* as a Super doesn't make it one, though. So when we call **`this.doit()`**, we go right back to the outside (OverridingSub) and start working our way in again, looking for a *doit* method. So the effect of invoking **`over.doitAgain()`** is the same as invoking *over*'s *doit*, *not* the Super method.

[Outside in pic]

10.2.3 Problems with Private

It isn't always completely straightforward to extend a class. Consider the BasicCounter and Resettable Counter classes from the chapter on Designing with Objects. Because the BasicCounter wasn't designed with inheritance in mind, there is a problem in extending it. In fact, we have to go back and modify the BasicCounter before we can describe the Resettable version directly in terms of it.

```
class BasicCounter implements Counting {

    int currentValue = 0;

    void increment() {
        this.currentValue = this.currentValue + 1;
    }

    int getValue() {
        return this.currentValue;
    }
}
```

To implement the Resettable Counter class, we would like to be able to write the following:

```
public class Counter extends BasicCounter implements Resettable {

    public Counter() {
        this.reset();
    }

    public void reset() {
        this.currentValue = 0;
    }
}
```

We have preserved the highlighting, and you can see that almost the entire new class is highlighted. This says that a Counter is just like a BasicCounter except:

1. It implements the Resettable interface (in addition to Counting, already implemented by — and hence inherited from — BasicCounter).

2. It has a no−args constructor that calls its own *reset* method.

3. It has a *reset* method that sets its *currentValue* field to 0.

But this code is not entirely adequate. In fact, it does not compile as is. The problem is that the currentValue field is not a part of the Counter class any more. The field *currentValue* is defined in BasicCounter. But BasicCounter's *currentValue* field is private, meaning that only BasicCounters (and the BasicCounter class, or factory) can access that field. The solution is to change the ***visibility*** of the field from private to ***protected***. This allows the Counter subclass to access BasicCounter's *currentValue* field. Now, the Counter code in this chapter does the same thing as the Counter code in the Chapter on Designing with Objects.

The moral here is that if you want your class to be extensible — to be able to be inherited from — you will need to make sure that subclasses can get access to anything that they need to be able to manipulate. This in turn opens those aspects of your class up to manipulation by other classes, since that information is no longer private. The visibility level *protected* is an intermediate point between private and public, but it does not always provide adequate protection. For details, see the chapter on Abstraction.

10.3 Constructors are Recipes

We already know that constructors give the special instructions for how to create a particular kind of object. How does this interact with inheritance?

10.3.1 The *this*() Expression

When a class has more than one constructor, we can express one constructor in terms of another using the special syntax this(). For example, we might define a Point class that either could be instantiated using specified values for the x and y coordinates or could take on the default value (0,0). We might define the constructors this way:

```
public class Point {

    private int x, y;

    public Point() {
        this(0, 0);
        // constructor would continue here....
    }

    public Point(int x, int y) {
        ...
    }
}
```

The line **`this(0, 0);`** in the first (no−args) constructor means “create me using my other constructor and the arguments 0, 0”. In other words, when we say **`new Point()`**, invoking the no−args constructor, this line transfers the responsibility of providing the instructions for the construction of the Point to the two−int constructor, supplying the ints 0 and 0 as values. Now, the second constructor would execute, creating a Point. This new Point's construction process would continue in the first constructor at the comment

```
// constructor would continue here....
```

The point being constructed would be the point resulting from the second constructor's invocation on 0, 0. Since there are in fact no more instructions in the first constructor after the comment, execution of this constructor would terminate and the new point returned would be the point corresponding to (0, 0).

The special buck−passing constructor this() can only be used as the first line of a constructor.

10.3.2 The *super*() Expression

Constructors and inheritance work similarly. Making an inherited object (the “inner object” that belongs to the superclass) is just like passing the buck to a same−class constructor. The first line of any constructor may be an explicit invocation of the superclass constructor, supplying whatever arguments are necessary between the parentheses.

For example, if we wanted to extend the CountingMonitor class, above, to determine whether the reading of its Counting had changed since the previous reading, we could add a field (to keep track of the previous reading) and a conditional in the *act* method. But how would we deal with the constructor? The beginning of this class might read:

```
public class ChangeDetectingCountingMonitor
                                extends CountingMonitor {

    private int previousReading;

    public ChangeDetectingCountingMonitor(Counting who) {
        super(who);
        // ...
    }
}
```

The first line of this constructor says “create my inner CountingMonitor instance using who as its constructor parameter.” When the superclass constructor completes its execution, the remainder of the ChangeDetectingCountingMonitor constructor body is executed, extending the CountingMonitor instance and wrapping it in whatever it needs to be a full−fledged ChangeDetectingCountingMonitor.

10.3.3 Implicit *super*()

We have seen that, when no explicit constructor is supplied, Java blithely inserts a no−args constructor. Java actually has two dirty little secrets about constructors:

1. **If no constructor is provided for a class, Java automatically adds a no−arguments constructor.**
2. **Unless a constructor explicitly invokes its superclass constructor or another (`this()`) constructor of the same class, Java automatically inserts `super();` as the first line of the constructor.**

This means that a class that doesn't seem to have a constructor actually has the following one:

```
public ClassName () {
    super();
}
```

What does this do? It means that you can create an instance of the class with **`new ClassName()`** — because the constructor has no parameters, so you don't have to give it any arguments — and it also means that each instance of *ClassName* has an instance of the superclass hiding inside it. That is, ***super();*** is a special incantation that means “Make me an instance of my superclass.” (Be careful: there are two readings of this request: “Give me an instance...” and “Turn me into an instance...”. The second reading is correct.)

The BasicCounter class has such an implicit, automatically inserted constructor, but the Counter class doesn't. Counter does automatically get the implicit call to **`super();`** though:

```
public BasicCounter () {
    super();
}
```

and

```
public Counter() {
    super();
    this.reset();
}
```

You can, of course, insert this no−args make−me−an−instance−of−my−superclass constructor into every class definition, and some people like to do so explicitly.

Details:

1. **`super();`** may only appear as the first line of a constructor.

2. The form **`super(args)`** may be used if the superclass constructor takes arguments.

3. If a constructor is defined, this constructor is not automatically added. So, for example, ***Echo*** *does not* have a no−args constructor.

4. If a superclass does not have a no−args constructor, an explicit call to **`super(args)`** must be used as Java's automatic insertion of **`super()`** will cause a compile−time error.

What if a class doesn't have a superclass? **Every class is a subclass except Object. If a class doesn't have an `extends` in its declaration, Java automatically inserts `extends Object`.** That means that the automatically−inserted constructor will in general make sense. **Beware:** Since Java will automatically invoke the no−args version of *super()* unless you explicitly invoke a superclass constructor, either (1) the superclass must *have* a no−args constructor or (2) you must explicitly invoke the superclass constructor yourself, supplying the requisite arguments. If you create a class without a no−args constructor, you can get into trouble extending it.

Style Sidebar

Explicit use of this. and super()

Although it is not strictly speaking necessary, it is good style to use **`this.`** wherever it is appropriate, i.e., to denote calls to an object's own fields or methods. While it makes your code somewhat more verbose, it also makes it easier to read and to understand what's going on. No method call should ever be made without reference to its target (i.e., whose method is being called). Field accessor expressions should always include a reference to the field's owner, distinguishing them from other name accesses (including parameter and local variable references).

A class declaration that does not contain an explicit `extends` clause still **`extends Object`**. Stating this explicitly may make it easier to read your code.

A constructor that does not call another (`this()`) constructor explicitly calls the superclass constructor. If the superclass constructor is not invoked explicitly, Java will insert an implicit call to *super()*, the superclass's no−args constructor. You can make this implicit call explicit by including **`super();`** as the first line of any constructor that doesn't explicitly invoke another self− or superclass constructor. This helps to remind you that it is being called anyway.

10.3.4 Multiple Views

Can have supertype, subtype, many along type hierarchy. Also interfaces. Can have multiple views simultaneously if multiple uses (names) have different types.

You all get the same behavior, no matter what type your reference is. Only difference is in what you can ask for.

10.4 Interface Inheritance

A class cannot inherit from an interface; it implements the interface, providing behavior to match the interface's specification. But one interface can extend another. Interface inheritance is much simpler than class inheritance. In interface inheritance, the methods and fields of the inherited (super) interface are simply combined into the methods and fields of the inheriting (sub) interface. The syntax for interface inheritance is identical to the syntax for class inheritance, but since there can be no overriding of method specifications, and since all fields are public and static therefore cannot be overridden, there is really no complexity to interface inheritance.

As with class inheritance, if one interface extends another, all instances implementing the

subinterface are instances belonging to both types.

10.5 Relationships Between Types

There are three different type−to−type relationships that will be important in creating systems. These three relationships correspond to three distinct mechanisms: implementation, extension, and coupling. ***Implementation*** is a relationship in which one type provides a specification and a second type provides a specific way of implementing that specification. In this case, the first type is called an interface and the second type is called a class. For example, an Alarm is one way of implementing the Resetable specification; an Animation is another. ***Extension*** is a relationship in which one type adds functionality to another. There are actually two variants of extension. In one, both types are specifications (i.e., interfaces) and the extending specification adds commitments to the extended specification. StartableAndResetable is an extension of Startable. In the other, both types are implementations (i.e., classes) and the extending implementation adds functionality to the extended implementation. A CheckingAccount adds check−writing functionality to a BankAccount. Extension is implemented using inheritance, the primary subject of this chapter. ***Coupling*** is a way of giving one object the ability to ask another to help it. For example, a MicrowaveOven may have a Clock, but a MicrowaveOven *isn't* a Clock. MicrowaveOven doesn't implement Clock behavior or extend it. Each MicrowaveOven *has* a corresponding Clock, and when the MicrowaveOven needs to know what time it is, it checks with its own Clock. In this case, the relationship is one−to−one (one MicrowaveOven per Clock, one Clock per MicrowaveOven). There are other cases in which the relationship may be many−to−one (many Chickens, one Coop) or one−to−many.

[Footnote: Unlike extension and implementation, coupling is really a relationship between instances; however, like implementation and extension, it is generally defined within the class.]]

It is important to know which of these three relationships ought to hold as you design your code.

It is always advisable to factor out common commitments and to separate the users of these contracts from their implementors. Wherever possible, an object should be known by an interface type rather than a class type to make it possible for alternate implementations to be used. This is true for both name declarations and method return types. The only time when an interface cannot be used routinely is in a construction expression.

[Footnote: But see, e.g., the Factory pattern [GHJV] for an approach to this problem.]

Interface implementation, the result of introducing these interfaces, is generally easy to recognize. An interface, after all, provides the contract without the actual implementation.

It is generally more difficult, especially for the novice programmer, to determine whether it is appropriate to use inheritance or merely containment. Inheritance is actually relatively rare (among classes) and should be used only when the new class really reuses the complete behavior of the existing class. This is because inheritance makes the implementation of the new class tremendously dependent on the details of the implementation of the existing class. Coupling is a much more general mechanism. In this case, the new kind of object simply relies on a previously existing kind of object to provide behavior, forwarding messages on to the instance of the pre−existing class. If the coupling relies on an interface type rather than on a class type, a different implementation can easily be substituted.

If you are constructing a class and want to make use of behavior implemented by another class, you must determine whether you are better off using inheritance (i.e., extension) or coupling. Here are some questions that you should ask:

- Does this new class present to its users the full range of behavior provided by the existing class (inheritance) or just some of that behavior (coupling)?
- Does this new class add behavior to the existing class (inheritance) or override it (coupling or a common subclass)?
- Can instances of this new class legitimately be treated as instances of the existing class (inheritance) or would this be inappropriate (coupling or common interface)?
- Does an instance of this new class have a different lifetimes from the associated instance of the existing class (coupling)?

It is only when the superclass will be wholly reused, and when the subclass really is an extension of the implementation provided by the superclass, that inheritance should be used. Occasionally, this justifies the use of an abstract class to encapsulate common behavior that is extended differently by different classes.

Abstract Classes

A class *can* have a method that is just a signature — an ***abstract*** method. In a class, however, the abstract method must be explicitly declared `abstract`. (Recall that methods in an interface are assumed to be abstract, even if they are not explicitly so declared.)

If a class has one or more abstract methods, it isn't a complete implementation. (It doesn't specify how to do the un−implemented method!) In this case you cannot directly make an instance of this class. (This is like a partial recipe — you can't cook anything edible with it, but it may be useful in building more complete recipes. We will see how to use one recipe to build another in the chapter on Inheritance.)

A class with one or more abstract methods is called an ***abstract class***. You cannot construct an instance of an abstract class.

[Footnote: Technically, a class can be abstract even if it has no abstract methods. However, *every* class with at least one abstract method *must* be declared `abstract`.]

Abstract classes can be useful when you want to specify a partial implementation. You should not use an abstract class when you only want to specify a contract; that is the function of an interface.

We will see examples of abstract classes in later chapters.

Chapter Summary

- Inheritance is a mechanism that allows one class to reuse the implementation provided by another.
- Inheritance should be used only when instances of the subclass can also reasonably be considered instances of the superclass.
- A class always extends exactly one superclass. If a class does not explicitly extend another, it implicitly extends the class `Object`.
- Method lookup always begins with an object's actual (most specific sub)class, even when the method is invoked by a `this.` expression in superclass code.
- A superclass method or (non−private) field can be accessed using a **`super.`** expression.
- If a constructor does not explicitly invoke another (`this()` or *super()* constructor, it implicitly invokes the superclass's no−args constructor.

Exercises

1. In the first interlude, we wrote “UpperCaser extends StringTransformer.” Explain.

2. Extend the Counter to count by 2.

3. Complete the definition of ChangeDetectingCountingMonitor from above.

4. In this exercise, you will re−implement AnimateTimer in two different ways and then compare them.

    a. Re−implement Timer by extending Counter.

    b. Extend the class in the previous exercise by making it Animate.

    c. Now re−implement AnimateTimer by extending AnimateObject directly.

    d. What if any type relations would exist between an instance of the class produced in (b) and the class produced in (c)?

Chapter 11

When Things Go Wrong: Exceptions

Chapter Overview

- What happens when something goes wrong?
- How do I create alternate ways to handle atypical circumstances?

This chapter covers mechanisms for dealing with atypical behavior. Sometimes, exceptional circumstances arise and require different mechanisms to cope with them. In this case, the normal entity−to−entity communication in your system may need to be interrupted. Java provides certain mechanisms for creating alternate paths through your community. These include the ***throw*** and ***catch*** statements as well as special ***Exception*** objects that keep track of these atypical circumstances.

This chapter includes sidebars on the syntactic and semantic details of `throw` and `catch` statements, Exception objects, and the requirement to declare exceptions thrown. It is supplemented by portions of the reference charts on Java methods and statements.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Objectives of this Chapter

1. To be able to read, understand, and write `throws` clauses as a part of interface and class contracts.

2. To learn how to `throw` and `catch` Exceptions and other Throwables.

3. To appreciate the role that anticipating exceptional circumstances plays in the design and testing of programs.

11.1 Exceptional Events

So far, the code that we have written has addressed "normal" situations in which nothing goes wrong. But sometimes, unusual things happen in our code, and we have to deal with them. In some cases, these unusual things are unexpected errors; in others, their existence is predictable but we may not know in advance when they are likely to happen. An example of this second kind is a network outage, which happens from time to time and can reasonably be anticipated, but is unexpected when it occurs. Planning for these ***exceptional circumstances*** and writing code that can cope with them is an important part of robust coding.

11.1.1 When Things Go Wrong

Consider the following example, drawn from the *StringTransformers* application of the first interlude. In that scenario, entities called *StringTransformers* are connected by "tin can telephone" entities called *Connections* . Each Connection has an end that you can put something into and an end that produces what you put into it. A StringTransformer can write to (or read from) a Connection if it is holding the appropriate end. In the user interface of that application, there is a way that a user can specify two StringTransformers to be connected. We are going to look in more detail at how the Connection actually gets attached to these two StringTransformers.

Let's say that the two transformers we're going to connect are *transformerA* and *transformerB*. In the code that is making the connection, we invoke the specific Connection constructor with these transformers as arguments:

```
new StringConnection(transformerA, transformerB)
```

The constructor code for StringConnection asks each of the transformers, in turn, to accept a(n input or output) connection. In fact, strictly speaking, *transformerA* and *transformerB* need not be StringTransformers at all; they need only implement the OutputAcceptor or InputAcceptor interfaces, since that is the only aspect of their behavior that we rely on here:

```
public StringConnection(OutputAcceptor a, InputAcceptor b) {
    a.acceptOutputConnection(this);
    b.acceptInputConnection(this);
}
```

This code is perfectly reasonable assuming that everything goes right. But what happens if *transformerA* already has an OutputConnection in place? It might be that *transformerA* is a Broadcaster or AlternatingOutputter or some other kind of transformer that can have many OutputConnections. It might be that *transformerA* is willing to throw away its existing OutputConnection and replace it with the one currently on offer. But it might also quite reasonably be that *transformerA* is unwilling and unable to accept an OutputConnection if it already has one in place. In this case, the StringConnection constructor code is in trouble.

This is precisely the sort of situation that we will deal with in this chapter. Something has gone wrong. We can anticipate in our design that this might happen. We want our code to respond appropriately. In other words, we want to design our programs to be able to handle exceptional circumstances.

11.1.2 Expecting the Unexpected

When you are designing a program, it is relatively easy to think about what is supposed to happen. You can act out the interactions that you want your program to have. You can draw out storyboards describing what comes next. You design interfaces and protocols to describe the roles each entity plays and the contracts it makes with others. But this is not enough.

In addition to figuring out what *ought* to happen, you also need to anticipate what *might* happen. That is, you need to understand what happens if a component does something unexpected; if the user does something foolish; if a resource that you depend on becomes unavailable or temporarily out of service; or even if a change that you make to your code inadvertently violates an assumption. In all of these cases, unexpected behavior of one portion of the system needs to be dealt with. Good design involves anticipating these possibilities and explicitly deciding what to do and designing for these circumstances.

Exceptional circumstances can be partitioned into three groups. One is the ***catastrophic failure***. In case of a catastrophic failure, there's really nothing that your program can do. This might happen, for example, if someone tripped over the power cord of the computer on which your program was running. In this case, it is reasonable to expect that your computer program will stop executing immediately. There's really nothing that you can do about a catastrophic failure.

[Footnote: At least at the time of failure. There are still things that you can do to plan for recovery from catastrophic failure. For example, a banking system may temporarily lose the functioning of an ATM, but it will not lose track of your bank balance entirely. It has been designed to keep this information safe even in the face of computer crashes.]

The second kind of exceptional circumstance is at the other end of the spectrum. This is a situation that is not the intended course of your program, but is so benign that it is dealt with almost as a matter of course. These are the unexpected situations that can be handled with a simple conditional or other testing mechanisms. For example, if we are about to perform a division operator, we might check to make sure that the divisor is not 0. In the extreme, these situations can be difficult to distinguish from “normal” operation.

Most exceptional circumstances fall between these two extremes. That is, they admit some intervention or even solution (unlike catastrophic failure), but handling these circumstances requires cooperation among entities or other additional complexity; it isn't possible or desirable to deal with this situation locally. These are the situations that you must take into account in your design.

When you are planning your program, you will be deciding how to partition the problem among a community of interacting entities and designing how these entities interact. At this point, you should also ask:

- What should happen if one of these entities is unreachable?
- What are all of the ways in which an entity might violate expectations?
- What should happen in each of these cases?
- What should an entity do if it has difficulty fulfilling its contract?

In each of these cases, you should decide whether the circumstance amounts to a catastrophic failure or can be handled by another entity. If it is a catastrophic failure, this circumstance ought to be documented; if not, it provides another set of interactions to build into your system. This ***exception handling*** becomes another part of your system design.

As you break each entity down — asking what is inside it, decomposing it into further communities of interacting entities — you should repeat these questions with respect to

these entities' mutual commitments. Eventually, you will decompose your problem to the level of individual operations and of interactions with entities outside the system that you are actually building. For these situations, you should ask:

- In what ways might this operation or outside entity fail?
- How else might it violate my expectations?
- Can I test for these circumstances prior to invocation of this operation or resource?
- What should I do if the failure or expectation violation occurs?

If the situation is one that can be ruled out using a simple test — such as checking for a zero divisor or verifying that the user's input is a legal value and asking for new input if not — such ***error checking*** should be introduced into your design. This strengthens the contracts that entities make with one another. Where violations cannot be handled locally, you will need to decide who should handle the issue and how it should behave.

11.1.3 What's Important to Record

At the time that an exceptional circumstance arises, the currently executing code is in the best position to determine what the problem is. It should take pains to record any information that might help other parts of the program (or a human user or debugger) figure out what happened. So, for example, in the case of a divide−by−zero error, it would be important to know what the expression was whose value was zero, causing the error. In the case of an invalid value entered by a user, it may be important to know what the invalid value is or what the legal values might be. It is also important to know what *kind* of thing went wrong: division by zero or illegal argument passed to a method or a label name that's `null` and shouldn't be or any of a whole host of possible values.

This information — what kind of thing went wrong and kind−specific additional information that might be useful for figuring out what the problem was or correcting it — is, in Java, encapsulated in a special kind of object. These are ***Exception*** objects. They signal what's gone wrong. There are many different (more specific) types of Exception objects, such as ***NullPointerException*** or ***IllegalArgumentException***. You can also define Exception types of your own (using inheritance). In addition to Exception, Java also defines a (similar but distinct) class of ***Error***s, meant to designate conditions of catastrophic failure, such as ***NoClassDefFoundError***. You can (but rarely will) define your own Errors as well.

Since Exceptions are objects, you can use them like any other object. If you define your own Exception classes, you can add any fields or methods that you think might be important to allow your program to handle the exceptional circumstance. One thing that is especially useful for an Exception to have is a String (suitable for printing to a user)

that explains something about what has gone wrong. In Java's Exception classes, such a String can be supplied to the constructor and retrieved using the instance's ***getMessage*** method.

It can also be very important to know where the problem occurred. Java's Exception classes record the point at which they were thrown (see below), but it can in addition be useful to record (e.g., in the message or in an additional field that you define) some program−specific indication of which code is reporting the exceptional circumstance and what it was trying to do when the exception occurred.

For example, in our OutputAcceptor code, we might recognize that we can't accept an OutputConnection if we already have one. In this case, we might create a new ConnectionRejectedException recording this circumstance:

```
new ConnectionRejectedException(this.toString()
                + " rejecting redundant OutputConnection")
```

The ConnectionRejectedException uses the *toString* method of the OutputAcceptor within which this code occurs to record who is rejecting the connection. An alternative is just to list the class name and method in a constant String:

```
"OutputAcceptor.acceptOutputConnection():  "
```

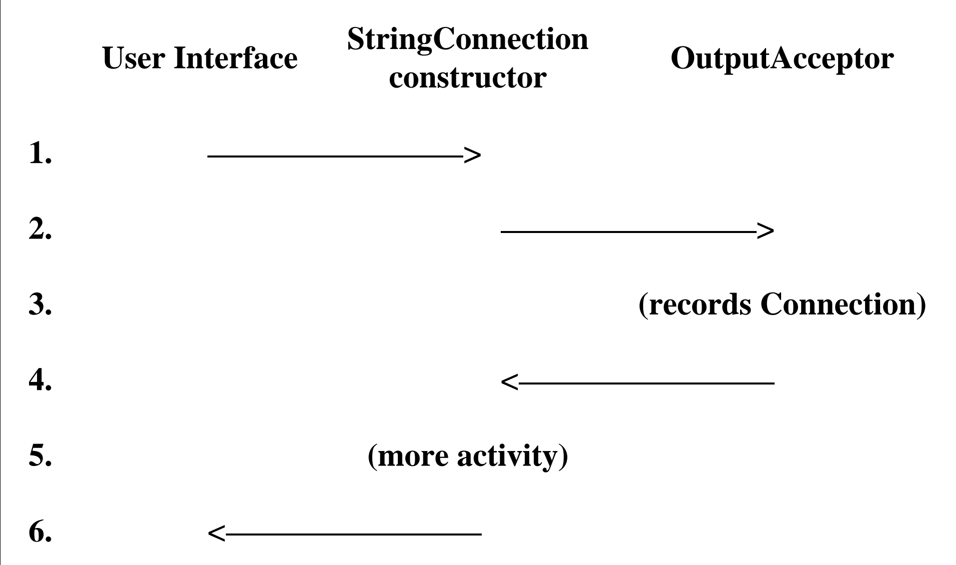
The ConnectionRejectedException might also record the existing OutputConnection and the newly supplied one; in the code fragment above, it does not do this.

Just defining a new exception isn't enough, though. Defining an exception is like composing a letter of complaint. In order for it to have any effect, you have to send out the letter. In the case of an Exception, this is accomplished by ***throwing*** the Exception.

11.2 Throwing an Exception

An Exception is an unusual circumstance that requires special handling. In order to understand how an Exception works — and what it means to throw one — we first need to look at how method invocation and return normally works.

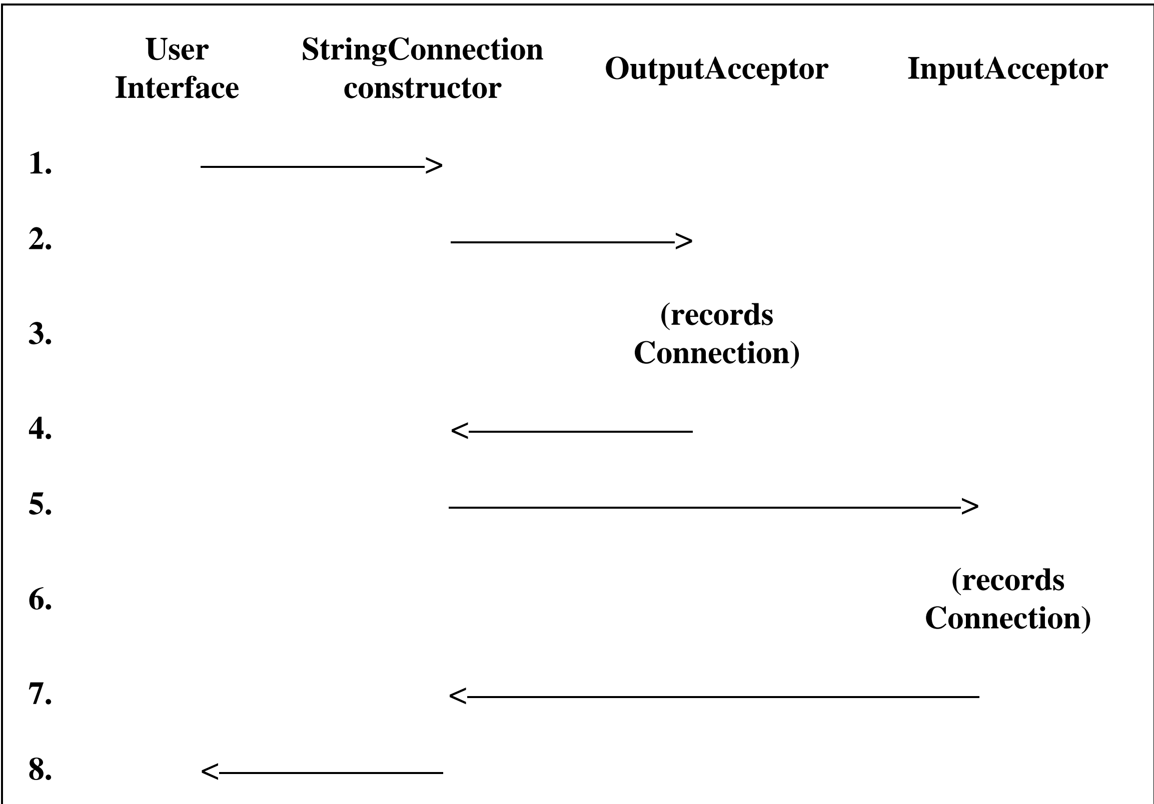
Let us begin by looking more closely at what happens in our new Connection example, when the user interface calls the StringConnection constructor, which in turn calls the OutputAcceptor's *acceptOutputConnection* method. We might diagram the normal control flow as follows:



The code from the User Interface invokes the StringConnection constructor, then the StringConnection constructor invokes the OutputAcceptor's *acceptOutputConnection* method. When the *acceptOutputConnection* method completes, it returns (nothing) to the StringConnection's constructor, which completes its work and provides the newly constructed StringConnection to the User Interface. These arrows are sometimes called the ***call path*** (and ***return path***) of this execution.

Communication among pieces of code is very simple. Each piece of code can only talk to the other pieces of code about which it knows. In this case, the User Interface knows about the StringConnection's constructor, and the StringConnection's constructor knows about the OutputAcceptor's *acceptOutputConnection* method. Think of it like an old−fashioned fire−fighting bucket brigade. All of the people line up from the water supply to the fire. A full bucket is passed from hand to hand down the line from the water supply to the fire. The empty bucket must be passed back the same way. In the normal motion of buckets, there is no way for a bucket to skip over a person; it must be passed from hand to hand, returning the way that it came.

[Footnote: In this example, the “more activity” line inside the constructor is a shorthand for a more complex picture. This “more activity” actually involves another method call, this one to the InputAcceptor's *acceptInputConnection* method. So the whole picture is more accurately represented as:



This doesn't violate the bucket brigade idea, but it does mean that the bucket brigade has a fork in it. The constructor can pass buckets to (i.e., invoke) both the OutputAcceptor's *acceptOutputConnection* method and the InputAcceptor's *acceptInputConnection* method.]

[Insert bucket brigade (throw versus normal method invocation) pic]

Throwing an Exception is different. What happens in this case looks more like the following:

	User Interface	**StringConnection constructor**	**OutputAcceptor**
1.	——————>		
2.		——————>	
3.			**OH NO!!**

When the OutputAcceptor's *acceptOutputConnection* method realizes that it has a problem it generates an Exception object, as we have seen above. Then, it throws the exception as hard as it can back the way it came. The Exception zooms back along the

call path, flying too fast to stop and execute any statements waiting for its return. In fact, the Exception keeps going until it encounters a compatible `catch` statement. If necessary, it may exit several method bodies. Or, if the `catch` is in the same block as the `throw`, it may not exit any method bodies at all. In other words, a `throw` statement sets an Exception flying, and the flying Exception can only be caught by a matching `catch` statement; no other intervening statement along the call path matters.

This is, in fact, just what we want. If the OutputAcceptor can't accept an output connection, we don't want the rest of the StringConnection's constructor to execute. For example, we don't want it to try to convince the InputAcceptor to accept the input end of the connection, because *this* connection isn't going to work out (since the OutputAcceptor isn't cooperating) and if the InputConnection accepts this one, then it won't later be able to accept a fully operational input connection. So when the OutputAcceptor decides that it has a problem, we want the Exception to propagate all the way back to the User Interface code, which should decide that connecting this particular pair of StringTransformers may not be such a good idea after all.

The code for the OutputAcceptor might look like this:

```
public class    ...    implements OutputAcceptor {

    private OutputConnection out;

    public void acceptOutputConnection(OutputConnection out) {
        if (this.out == null) {
            this.out = out;
        } else {
            throw new ConnectionRejectedException(
                         this.toString()
                            +
                       " rejecting redundant OutputConnection");
        }
    }
}
```

This example introduces a new statement type, ***throw***, and a new declaration element, ***throws***. (Note the *s* on the declaration element.) The ***throw*** statement works just as we have described; it abruptly terminates the execution of this method and causes the Exception to propagate backwards along the return path until a compatible `catch` statement is encountered. (We will see this below.)

What about the ***throws*** clause? ***Throwing an exception*** is actually part of the contract that one object makes with another. It is as much a part of a method's contract as its (normal) return type or the parameters it needs. So a method must declare that it may throw an exception (and what type of exception it may throw). This way, anyone calling the method knows to be prepared for it to throw this exception. The `throws` clause is

the final part of a ***method signature***, and `throws` clauses may appear in interface (abstract) method declarations as well as in method definitions.

Throws clauses are not restricted to methods. Constructors, too, must declare any exceptions that they throw. A constructor can explicitly throw an exception using a `throw` statement. A constructor (or method) can also throw an exception by calling something that throws an exception and then not catching it. This is what happens with the StringConnection constructor. Here it is, reprinted from above, with the added `throws` clause italicized.

```
public StringConnection(OutputAcceptor a, InputAcceptor b)
                      throws ConnectionRejectedException {
    a.acceptOutputConnection(this);
    b.acceptInputConnection(this);
}
```

The StringConnection constructor invokes OutputAcceptor's *acceptOutputConnection method. If the OutputAcceptor doesn't accept the output connection, the StringConnection constructor isn't going to be able to fix this. So the StringConnection constructor should itself exit abruptly. In other words, the Exception thrown by acceptOutputConnection flies right out of the StringConnection constructor as well, still waiting to find a compatible* `catch` *clause.*

`Throw` Statements and `Throws` Clauses

A ***throw*** statement looks a lot like a ***return*** statement, but it always takes an argument (which can be in parentheses or not), and its argument must be something legal to throw. Anything that **`extends Throwable`** is legal to throw. In particular, this includes anything that **`extends Exception`**.

The effect of a `throw` statement is that execution abruptly returns up the call path until a compatible ***catch*** clause is encountered. Nothing except a compatible `catch` clause can stop the propagation of a thrown object.

If an Exception (except a RuntimeException) is thrown and not caught within a method or constructor body, you must also declare that that method or constructor ***throws*** the Exception. This is a part of the ***signature***, like saying what a method returns or what arguments a method or constructor expects.

The `throws` clause appears after the argument list, but before the method/constructor body. The syntax for a `throws` clause is

```
throws ExceptionType1, ExceptionType2, ... ExceptionTypeN
```

Every exception thrown and not caught within the body must match (at least) one of the exception types declared thrown by the method or constructor. If the method or constructor throws only a single exception type, the list contains no commas.

11.3 Catching an Exception

We have seen how an exception can be generated and thrown. We have also seen that a thrown exception keeps flying until it encounters a compatible `catch` statement. Now, we will look at `catch` statements and how they work. This code introduces new syntax: the ***try/catch*** statement type. If `throws` is syntactically like `return, then try/catch is a bit like if/else.`

A `catch` statement is properly a `try/catch` statement (or even more properly a `try/catch/finally` statement). If you are about to execute a statement that might throw an exception that you'd like to catch, you must first enter a ***try block***. This is just like a regular block, except that it is preceded by the Java keyword `try`. This notifies Java that exceptions may be thrown and that it should be on the lookout for the ones that you want to catch.

At the end of the possible−exception−throwing code, you end the `try` block and introduce a `catch` clause. A ***catch clause*** contains a parameter declaration of the type that you wish to catch. The `catch` clause has a block that describes the instructions to execute if one of these is caught.

For example, the code in the User Interface that is trying to connect *transformerA* (here named by *to*) and *transformerB* (here named by *from*) might say:

```
try {
    new StringConnection(to, from);
} catch (ConnectionRejectedException e) {
    Console.println("Sorry, can't make that connection."
                       + "  Please try again.");
}
```

- This `try/catch` statement type has two bodies: one after the keyword `try`, and one after the `catch` parameter.

- The `try` body is a statement or set of statements that may throw an exception. In this case, we know that the StringConnection constructor may throw ConnectionRejectedException. We can tell this from its declaration, and so can the Java compiler.

- The `catch` portion of the statement has a single parameter, the exception that is to be caught, written in the usual *Type−of−thing Name−of−thing* syntax. In this case, the exception type is ConnectionRejectedException, and the name of the exception is *e*. The name is required, and it may be used inside the `catch` body, just like a method parameter name can be used inside the method body. It is common to name the exception *e*, though there's no particular reason for it; it's just like loop variables are often named *i*.

- The `catch` body contains statements which are executed if and only if the appropriate type of exception is thrown. (The "appropriate type" is the type of the `catch`'s parameter.) Inside the `catch` body, the parameter name may be used to refer to the exception, though there isn't a *whole* lot you can do with an exception other than print its message.

In this case, once the exception is caught, a message is printed to the user. This statement might itself appear inside an animate object's *act* method, so that something is continually listening to the user and trying to make connections on the user's behalf. This message lets the user know that this particular attempt didn't work. If we had supplied additional information along with the exception, we might use it at this point to give the user more information (perhaps flashing the object that refused the connection) or to try to repair the situation (asking whether the user means to delete the existing connection, for example,

and then retrying the connection creation).

One `try` can actually have several `catch` statements. In this case, once something is thrown inside the `try` body, it is compared against the `catch` parameter statements in order until one that matches is found. If a match is found, only the first matching `catch` body is executed; then control continues at the end of the `try/catch` statement. If no match is found, the thrown object continues exiting statement blocks until a corresponding `catch` is found.

For completeness's sake, it is worth mentioning that a `try/catch` statement can have a ***finally*** clause (so that it's really ***try{}catch(){}finally{}***). In this case, no matter how the statement is exited — regardless of whether something is thrown, and regardless of whether the thrown object is caught — the `finally` statement *will* be executed. At this point, you shouldn't need to be using `finally`, but if you ever need to know, the gory details are included in the Java language specification.

`Try` Statement Syntax

A ***try/catch/finally*** statement has a body after ***try***, a body after each ***catch*** clause, and a body after the ***finally*** clause if it is present. Each of these bodies is a normal block executed according to the usual block execution rules. If a `catch` block is executed (i.e., if a matching Throwable has been caught), the `catch` parameter is bound to the caught object during execution of the `catch` block.

The ***try body*** is a statement or set of statements that may throw an exception. Although not every execution of the `try` statement must throw an exception, the `try` statement must contain at least one expression that is declared as throwing each of the types of exceptions listed in its `catch` clauses.

Each ***catch clause*** has a single parameter (type and name) followed by a block. A ***catch clause matches*** the thrown object exactly when the thrown object can be named by a name of the catch clause's parameter type. Only the first matching catch clause is executed.

The `try` statement is executed as follows:

- The `try` block is executed in order until something is thrown or the end of the `try` body is reached.
- If nothing is thrown during the `try` body, execution continues after the final `catch` clause of the `try/catch` statement.

- If something is thrown during the `try` body, it is compared against the parameter of each `catch` block, in turn, until a match is found. In this case, that `catch` block is executed (as a normal block) with the parameter bound to the (matching) caught object. At most one `catch` block of a `try` statement is executed.

A `try` statement may also have a single optional ***finally clause***. This is the keyword `finally` followed by a block. If the `try` statement is entered, the `finally` clause is always executed. This leads to somewhat complicated execution rules, described below and further documented in Java's language specification. Keyword `finally` clauses are largely outside the scope of this book and are included here only for completeness.

The following two points explain the special behavior of `try` statements with `finally` blocks:

- After execution of at most one matching `catch` block, execution proceeds at the `finally` block (if it is present). If a `try` statement is entered, its `finally` block is always executed, regardless of the execution within the `try` statement.

- If no uncaught exceptions remain on exiting the `finally` block, execution proceeds after the end of the `try/catch/finally` statement. If there is an outstanding thrown object, execution proceeds with the continued flight of that Throwable.

11.4 `Throw` versus `Return`

There are both similarities and differences between `throw` and `return` statement types. Both involve a single Thread following instructions that may take it from one method or constructor to another, often moving across multiple objects. From the perspective of the Thread , the objects (and their methods and constructors) are providing roles that it plays, scripts that it reads, or instructions that it follows.

When a Thread is executing some instructions and reaches a method invocation expression (or an instance creation expression), it carefully records its current place in the script, puts the current script down on the table in front of it, and picks up the invoked method script. If fulfilling that expression in turn involves a further invocation, yet another script will be added to the pile on the table. When an invocation completes, the Thread puts the corresponding script away and returns to the carefully marked pending method invocation (or instance creation) expression on top of the pile.

In other words, to clear off the pile, the Thread must pick up each script in order on its way out and complete any remaining instructions before going on to the next. Every method invocation or instance creation expression eventually returns control to the body of code from which the call was invoked. The Thread eventually returns to the carefully marked spot and continues from there.

A major difference between `return` and `throw` statements is in how this execution proceeds, i.e., whether the Thread continues executing one instruction at a time or simply flies over the instructions looking for a matching `catch` statement. When a Thread returns normally from a method, execution continues one instruction at a time. When a Thread encounters a `throw` statement, it steps back through its pile of carefully marked scripts rather rapidly, scanning down the instructions until an appropriate `catch` statement is encountered. If the current script doesn't contain a matching `catch` statement, it is summarily discarded and the next script is examined in turn.

This means that a `return` statement always causes the current method to complete, returning control to whomever called this method. This is true no matter how many statement blocks the return is buried inside. **A `return` always exits exactly one method invocation.**

In contrast, a `throw` exits one *block* at a time until a `catch` of the appropriate type is found. This means that a `throw` may not exit any methods (if the `throw` occurs directly inside an appropriate `try/catch`), or the `throw` may exit many methods (if the exception is not caught in any of these calling methods). **A `throw` exits blocks until an appropriate `catch` is encountered.**

Exceptions, Errors, and RuntimeExceptions

In Java, any instance whose class extends the class ***Throwable*** can be thrown and caught. Two special subclasses of Throwable are defined for use under exceptional circumstances:

- ***Error*** is the Java class that denotes a catastrophic failure. This is an event from which your program is not expected to be able to recover. A well−designed robust program that is expected to have an extended lifetime (such as a banking system or an airline reservation system) must have ways of dealing with catastrophic failure, but most programs that you write will not have to worry about such circumstances.

- ***Exception*** is the Java class that indicates a non−catastrophic failure. Exceptions are circumstances from which your program should recover (or at least exit gracefully).

All Java built−in Error and Exception classes have two constructors, one that takes no arguments and one that takes a single String argument. This String can be accessed using the ***getMessage*** method of Error or Exception. If you define your own subclass, it is a good idea to define these two constructors there as well.

RuntimeException is a special subclass of Exception. RuntimeExceptions are circumstances from which your program should recover, but — unlike for other Exceptions — methods and constructors throwing RuntimeExceptions do not have to declare this fact.

All Exceptions other than RuntimeExceptions are called ***checked exceptions***. A method or constructor that may throw a checked exception must declare this fact, allowing the compiler to check for the presence of exception handlers. This can be very helpful in debugging, so you will generally want to extend Exception rather than RuntimeException.

When overriding a superclass method, a subclass method may only throw those checked exceptions also declared by the (overridden) superclass method. In other words, an overriding method may throw fewer things than promised by its superclass, but it may not throw additional (checked) things.

11.5 Designing Good Test Cases

One of the most important parts of being a good programmer is knowing how to test your code. To begin this phase, write down all of the assumptions that your code makes. Think of something that violates one of these assumptions; will this break your code? How

about something that violates three of these assumptions?

Once you have all of your assumptions written down, think about things that are extreme but within your assumptions. Try to design test cases for these. Think of every feature your code has, and every situation in which this feature could possibly be exercised. Design test cases for these as well. And don't forget the simple cases; it is always worth testing these as well as the pathological ones.

Your goal should be to test your code thoroughly and exhaustively, so you should design your test suite to exercise your program as fully as possible. You should also design test cases to catch bugs you think that other people might make. In particular, you should try to identify any weaknesses or difficult cases and design examples that stress these elements.

Finally, you should keep this test suite around, so that as you modify your code, you can test it again on these same examples, making sure that it still handles all of the old cases. This is called ***regression testing***.

Chapter Summary

- In designing a program, you should anticipate things that can go wrong and design in mechanisms to deal with them.
    - ♦ Catastrophic failures cannot be prevented, but certain systems need to design in mechanisms to minimize the damage that they cause.
    - ♦ Some failures can be anticipated and avoided through simple checks and guards.
    - ♦ Other failures must be handled as they arise, often using Java's exception handling mechanisms.
- Exceptions should record information that is useful for addressing the problem as well as information that is useful for advising the debugger or the human user.
- When an exception is thrown by a method or constructor, it exits each enclosing block in turn until a matching `catch` statement is encountered.
- Methods and constructors that may throw checked exception types must declare this fact in their signatures.
- A method invocation or constructor that may throw a checked exception may be safely invoked within a `try` block with a corresponding `catch` statement. The `catch` statement is responsible for attempting to recover from the exception.

Exercises

1. Describe the process of baking a cake. Include at least three exceptional circumstances that might arise and how these should be handled.

2. Describe the normal conduct of a soccer game. Include at least three exceptional circumstances that might arise and how these should be handled.

3. Define an Exception type called UnbelievableException. Remember to define two constructors.

4. Using your UnbelievableException type, write an animate object that continually asks the user for the user's age, then throws an UnbelievableException if appropriate. Note: the presence of an unbelievable age should *not* cause the program to terminate.

Part 4

Refining Interactions

Chapter 12

Dealing with Difference: Dispatch

Chapter Overview

- How can I do different things at different times or under different circumstances?
- How can one method respond appropriately to many different inputs?

In previous chapters, we have looked at entities that respond to each input in roughly the same way. In this chapter, we will look at how an entity can respond differently depending on its input. In particular, we will look at how to build the ***central control loop*** of an entity whose job is to ***dispatch control*** to one of a set of internal ***"helper" procedures***.

This chapter introduces several mechanisms for an entity to generate different behavior under different circumstances. ***Conditionals*** allow you to specify that a certain piece of code should only be executed under certain circumstances. This allows you to prevent potentially dangerous operations — such as dividing by zero — as well as to provide variant behavior.

The decision of how to respond often depends on the value of a particular expression. If there are a fixed finite number of possible values, and if the type of this expression is integral, we can use a special construct called a ***switch statement*** to handle the various options efficiently. A `switch` statement is often used together with ***symbolic constants***, names whose most important property is that each one can be distinguished from the others.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Arrays are specialized collections of things. They allow you to treat a whole group of things uniformly. Arrays can be used to create conditional behavior under certain circumstances.

Procedural abstraction (covered in the next chapter) also plays a crucial role in designing good dispatch structures.

This chapter includes sidebars on the syntactic and semantic details of `if`, `switch` , and `for` statements, arrays, and constants. It is supplemented by portions of the reference chart on Java Statements.

Objectives of this Chapter

1. To understand what *dispatch* is, and its relationship to the central control loop.

2. To understand several mechanisms for dispatch: `if-else` statements, `switch` statements, and arrays.

3. To know when to choose which mechanism, and why.

4. To appreciate the value of symbolic constants.

5. To gain a deeper understanding of conditionals (e.g., by looking at cascaded conditionals).

6. To gain a deeper understanding of loops (e.g., by looking at `for` loops).

7. To understand arrays: how to declare them, define them, and use them for simple iteration and for dispatch.

12.1 Conditional Behavior

The animate objects that we have seen so far generally execute the same instructions over and over. A clock ticks off the time. A StringTransformer reads a string, transforms it, and writes it out. A web browser receives a url request, fetches, and displays the web page. And so on. These entities repeatedly execute what we might call a ***central control loop***, an infinitely repeated sequence of action.

In this chapter, we look instead at entities whose responses vary from one input to the next, based on properties of that input. The actual responses are not the subject of this chapter; instead, we will largely assume that the object in question has methods to provide those behaviors. The topic of this chapter is how the central control loop selects

among these methods. This function — deciding how to respond by considering the value that you have been asked to respond to — is called ***dispatch***.

Imagine that we are building a calculator. One part of the calculator — its ***graphical user interface***, or ***GUI*** — might keep a list of the buttons pressed, in order. The central controller might loop, each time asking the GUI for the next button pressed. The primary job of this central control loop would be to select the appropriate action to take depending on what kind of button was pressed, and then to dispatch control to this action−taker.

For example, when a digit button is pressed, the calculator should display this digit, perhaps along with previously pressed numbers.

[Footnote: Pressing 6 right after you turn on a calculator is different from pressing 6 after pressing 1 right after you turn on a calculator. In the first case, the calculator displays 6; in the second, it displays 16.]

Pressing an arithmetic function key — such as + or * — means that subsequent digits should be treated as a new number — the second operand of the arithmetic operator — rather than as additional digits on the first. Pressing = causes the calculator to do arithmetic. And so on.

In this example, the calculator's central control loop is behaving like a ***middle manager***. It's not the boss, who gets to set direction. It's not the worker, who actually does what needs to be done. The dispatcher is there to see that the boss's directions (the button pressed) get translated into the appropriate action (the ***helper procedure***). The dispatcher is simply directing traffic. This kind of behavior, in which different things happen under different circumstances, requires ***conditional behavior***. We have already seen a simple kind of conditional behavior using Java's `if` statement. In this chapter, we explore several different means of achieving conditional behavior in greater detail.

Throughout this chapter, we will assume that we have methods that actually provide this behavior. For example, the calculator might have a *processDigitButton* method which would react appropriately when a number key is pressed. Another method, *processOperatorButton*, would apply the appropriate operation to combine the value currently showing on the calculator's display with the number about to be entered. We will also use methods such as *isDigitButton* to test whether a particular *buttonID* corresponds to a number key. Separating the logic surrounding the use of these operations from their implementation is an important part of good design and the topic of much of Chapter 13, *Encapsulation*.

In this chapter, we are going to concern ourselves with what comes after the first line of the calculator's *act* method:

```
public void act() {
    SomeType buttonID = this.gui.getButton();
    ...
}
```

The remainder of this method should contain code that calls, e.g., *processDigitButton* if *buttonID* corresponds to one of the buttons for digits 0 through 9, or *processOperatorButton* if *buttonID* corresponds to the button for addition. This chapter is about deciding which of these is the correct thing to do.

12.2 Keywords *if* and *else*

We have already seen the `if` statement, Java's most general conditional. Almost every programming language has a similar statement type. An ***if statement*** is a compound statement involving a ***test expression*** and a body that can include arbitrary statements. Any conditional behavior that can be obtained in Java can be accomplished using (one or more) `if` statements. An `if` statement corresponds closely to normal use of conditional sentences in every−day language. For example, “If it is raining out, take an umbrella with you” is a sentence that tells you what to do when there's rain. Note that this sentence says nothing about what to do if there is no rain.

12.2.1 Basic Form

Every `if` statement involves two parts: the ***test expression*** and the ***consequent statement***. The test expression represents the condition under which the consequent should be done. The test expression is some expression whose type *must* be `boolean`. In our example sentence, this boolean expression is “it is raining out.” This expression is either true or false at any given time,

[Footnote: Excluding that sort of grey dreary drippy weather that haunts London and certain times of the year in Maine, of course.]

making it a natural language analog to a true−or−false boolean. In Java, this expression must be wrapped in parentheses.

When an `if` statement is executed, this conditional expression is ***evaluated***, i.e., its value is computed. This value is either `true` or `false` . The evaluation of the boolean test expression is always the first step in executing an `if` statement. The rest of the execution of the `if` statement depends on whether this test condition is `true` or `false` .

In the English example above, if “it is raining out” is true — i.e., if it is raining out at the time that the sentence is spoken — then you should take an umbrella with you. That is, if the condition is true, you should do the next part of the statement. This part of the `if` statement — the part that you do if the test expression's value is `true` — is called the

consequent.

In Java, execution of an `if` statement works the same way. First, evaluate the boolean test. If the value of the test expression is `true`, then execute the consequent. If the value of the test expression is `false` , the consequent is not executed. In this case, evaluating the test expression is the only thing that happens during the execution of the `if` statement. Note that the value of the expression that matters is its value *at the time of its evaluation*. If the test is executed at two different times, it may well have two different values at those times.

In Java, the consequent may be any arbitrary statement (including a block). In this book, we will always assume that the consequent is a block, i.e., a set of one or more statements enclosed in braces.

[Pic of if execution path]

We could write pseudo−code for our English conditional as follows:

```
if (currentWeather.isRaining()) {
    this.take(umbrella);
}
```

This isn't runnable code, of course, but it does illustrate the syntax of a basic `if` statement: the keyword `if`, followed by a boolean expression wrapped in parentheses, followed by a block containing one or more statements. To execute it, we would first evaluate the (presumably boolean) expression

```
currentWeather.isRaining()
```

(perhaps by looking out the window) and then, depending on whether it *is* raining, either take an umbrella, i.e., execute

```
this.take(umbrella)
```

or skip it.

A somewhat more realistic example is the following code to replace a previously defined number, *x*, with its absolute value:

```
if (x < 0) {
    x = - x;
}
```

This code does nothing just in case *x* is greater than or equal to 0.

[Footnote: It evaluates the expression **`x < 0`**, of course, but it "does nothing" that has any lasting effect.]

If *x* happens to be less than 0, the value of *x* is changed so that *x* now refers to its additive inverse, i.e., its absolute value.

Note that the same `if` statement may be executed repeatedly, and the value of the boolean test expression may differ from one execution of the `if` statement to the next. (For example, it may be raining today but not tomorrow, so you should take your umbrella today but not tomorrow.) The value of the boolean test expression is checked exactly once each time the `if` statement is executed, as the first step of the statement's execution.

12.2.2 The *else* Keyword

The `if` statement as described above either executes its consequent or doesn't, depending on the state of the boolean test expression at the time that the `if` statement is executed. Often, we don't want to decide whether (or not) to do something; instead, we want to decide which of two things to do. For example, if it's raining, we should take an umbrella; otherwise, we should take sunglasses. We could express this using two `if` statements:

```
if (currentWeather.isRaining()) {
    this.take(umbrella);
}

if (! (currentWeather.isRaining())) {
    this.take(sunglasses);
}
```

Recall that `!` is the Java operator whose value is the boolean opposite of its single argument. So if

```
currentWeather.isRaining()
```

is true, then

```
! (currentWeather.isRaining())
```

is false; if

```
currentWeather.isRaining()
```

is false, then

```
! (currentWeather.isRaining())
```

is true.

These two conditional statements, one after the other, are intended to express alternatives. But they don't, really. For example, the two statements each check the boolean condition **`currentWeather.isRaining()`**. This is like looking out the window twice. In fact, the answer in each of these cases might be different. If we don't get around to executing the second `if` statement (i.e., looking out the window the second time) for a little while, the weather might well have changed and we'd find ourselves without either umbrella or sunglasses (or with both). The weather doesn't usually change that often (except in New England), but there are plenty of things that your program could be checking that do change that quickly. And, since your program is a community, it is always possible that some other member of the community changed something while your back was turned.

[Footnote: But see chapter 20, *Synchronization*, where we discuss mechanisms to prevent the wrong things from changing behind your back.]

Instead of two separate `if` statements, we have a way to say that these two actions are actually mutually exclusive alternatives. We use a second form of the `if` statement, the `if/else` statement, that allows us to express this kind of situation. An ***if/else*** statement has a single boolean test condition but two statements, the ***consequent*** and the ***alternative***. Like the consequent, the alternative can be almost any statement but will in this book be restricted to be a block.

[Pic of if/else execution path]

Executing an `if/else` statement works mostly like executing a simple `if` statement: First the boolean test expression is evaluated. If its value is true, the consequent statement is executed and the `if/else` statement is done. The difference occurs when the boolean test expression's value is false. In this case, the consequent is skipped (as it would be in the simple `if`) but the alternative statement is executed in its place. So in an `if/else` statement, exactly one of the consequent statement or the alternative statement is *always* executed. Which one depends on the value of the boolean test expression.

The following code might appear in the calculator's *act* method, as described above. It is looking at which button is pressed, just like a good manager, and deciding which helper procedure should handle it.

```
if (this.isDigitButton(buttonID)) {
    this.processDigitButton(buttonID);
} else {
    this.processOperatorButton(buttonID);
}
```

This code presumes some helper functions. The method *isDigitButton* verifies that the *buttonID* corresponds to the keys 0 through 9. The *process...* methods actually implement the appropriate responses to these button types.

Because there is only one test expression in this statement, it is always the case that at the single time of its evaluation (per `if` statement execution), it will be either true or false. If the test expression is true, the consequent statement will be executed (and the alternative skipped). If it is false, the alternative statement will be executed (and the consequent skipped). Exactly one of the consequent or the alternative will necessarily be executed each time that the `if` statement is executed.

12.2.3 Cascaded *if* Statements

The `if/else` statement is a special case of a more general situation. Sometimes, it is sufficient to consider one test and decide whether to perform the consequent or the alternative. But the example we gave of determining whether the *buttonID* was a digit or not probably isn't one. After all, a non−digit might be an operator, but it also might, for example, be an `=` . We probably need to check more than one condition, although we know if any one of these conditions is true, none of the others is. This is a perfect situation for a ***cascaded if*** statement.

[Footnote: The test for *isDigitButton*, etc., may seem mysterious right now, and indeed we will simply assume the existence of these boolean−returning predicates for now. An implementation is provided in the section on Symbolic Constants, below, and discussed further in Chapter 13, *Encapsulation*.]

```
if (this.isDigitButton(buttonID)) {
    this.processDigitButton(buttonID);
} else {
    if (this.isOperatorButton(buttonID)) {
        this.processOperatorButton(buttonID);
    } else {
        this.processEqualsButton(buttonID);
    }
}
```

In fact, the situation is really even more complex:

```
if (this.isDigitButton(buttonID)) {
    this.processDigitButton(buttonID);
} else {
    if (this.isOperatorButton(buttonID)) {
        this.processOperatorButton(buttonID);
    } else {
        if (this.isEqualsButton(buttonID)) {
            this.processEqualsButton(buttonID);
        } else {
            // and so on until...
            throw new NoSuchButtonException(buttonID);
        }
   }
}
```

These `ifs` inside `elses` can get to be quite difficult to read, not to mention the pressure that they put on the right margin of your code as each subsequent `if` is further indented.

[Footnote: The final lines of such a sequence also contain an awful lot of closing braces.]

In order to avoid making your code too complex — and too right-handed — there is an alternate but entirely equivalent syntax, called the ***cascaded if*** statement. In this statement, an `else` clause may take an `if` statement directly, rather than inside a block. Further, the consequent block of this embedded `if` statement is lined up with the consequent block of the original `if` statement. So the example above would now read:

```
if (this.isDigitButton(buttonID)) {
    this.processDigitButton(buttonID);
} else if (this.isOperatorButton(buttonID)) {
    this.processOperatorButton(buttonID);
} else if (this.isEqualsButton(buttonID)) {
    this.processEqualsButton(buttonID);
    // and so on until...
} else {
    throw new NoSuchButtonException(buttonID);
}
```

Note that instead of ending with many close braces in sequence, a cascaded `if` statement ends with a single `else` clause (generally without an `if` and test expression) followed by a single closing brace.

[Pic of cascaded if execution path]

Like a simple `if/else` statement, exactly one block of a cascaded `if` statement is executed. Once that block executes, the entire statement is finished. The difference is that if the first expression's value is false, the next condition is evaluated, and then the next, and so on, until either

- one test expression evaluates to true, in which case the corresponding body is executed and execution of the statement is then terminated, or
- an `else` without an `if` and test is reached, in which case the corresponding body is executed, or

- the end of the statement is reached, in which case its execution is complete.

Since an `else` with no `if` and test is always executed, such an `else` must be the last clause of the cascaded `if`.

12.2.4 Many Alternatives

A conditional is a very general statement. With it, it is possible to write extremely convoluted programs. In order to make your program as easy to understand as possible, it is a good idea to keep your conditionals clean. A reasonable rule of thumb is that you should be able to explain the logic of your `if` statement easily to a friend. If you have to resort to pen and paper, your conditional expression may be too complex. If you have to write down more than two or three things, your ***conditional logic*** is most likely out of control.

For example, you should not test too many things simultaneously in one test expression. If you have a complex condition to test, use a boolean−returning method (a ***predicate***) to keep the test expression simple. By naming the predicate appropriately, you can actually make your code much easier to read, as we did with *isDigitButton* and *isOperatorButton*, above. We will return to this point in the section on Procedural Abstraction in Chapter 13, *Encapsulation*.

As we have seen, you can embed `if` statements. In the example that we gave above, the embedded statements were actually mutually exclusive alternatives in the same set of tests: the button is either a digit or an operator or the equals button or.... In this case, you should use the cascaded `if` syntax with which we replaced our embedded `if`s.

But sometimes it is appropriate to embed conditionals. For example, in the calculator's *act* method, inside the *isOperatorButton* block, we might further test whether the operation was addition or subtraction or multiplication or division.

```
if (this.isDigitButton(buttonID)) {
    this.processDigitButton(buttonID);
} else if (this.isOperatorButton(buttonID)) {
    if (this.isPlusButton(buttonID) {
        this.handlePlus();
    } else if (this.isMinusButton(buttonID)) {
        this.handleMinus();
    } else if (this.isTimesButton(buttonID)) {
        this.handleTimes();
    } else if (this.isDivideButton(buttonID)) {
        this.handleDivide();
    } else {
        throw new NoSuchOperatorException(buttonID);
    }
} else if (this.isEqualsButton(buttonID)) {
    // etc.
}
```

In this case, these further tests are a part of deciding how to respond to an operator button, including an operator−specific exception−generating clause. Note that the additional tests appear inside an `if` body, not inside an unconditional `else`. Using an embedded conditional to further refine a tested condition is a reasonable design strategy. **Beware:** of ***multiply evaluating an expression*** whose value might change. Instead, evaluate the expression once, assigning this value to a temporary variable whose value, once assigned, will not change between repeated evaluations.

The example above of looking out the window to check the weather may work well in southern California, but it is ill−advised in New England, where the weather has been known to change at the drop of a hat. Similarly, ***repeated invocation of a method*** returning the current time can be expected to produce different values. So can repeated invocations of a Counting's *getValue* method. If we execute the following conditional:

```
if (theCounter.getValue() > 1) {
    Console.println("My, there sure are a lot of them!");
} else if (theCounter.getValue() == 1) {
    Console.println("A partridge in a pear tree!");
} else if (theCounter.getValue() == 0) {
    Console.println("Not much, is it?");
} else if (theCounter.getValue() < 0) {
    Console.println("I'm feeling pretty negative.");
} else {
    Console.println("Not too likely, is it?");
}
```

while *theCounter* is independently incremented, it is possible that the counter will be incremented in just such a way that "Not too likely" might be printed. **Question:**

Describe how the process of executing this conditional (once) might be intertwined with the (repeated) incrementing of the counter to result in any one of the five different strings being printed, including the last possibility. (So, for example, describe how "My, there sure are a lot of them!" might get printed, how "A partridge in a pear tree!" might get printed, and so on.) Is it possible that nothing is printed? Is it possible that more than one string is printed?

If/Else Statement Syntax

An ***if statement*** consists of the following parts:

- The keyword `if`, followed by
- an expression of type `boolean`, enclosed in parentheses, followed by
- a (block) statement.

This may optionally be followed by an `else` clause. An ***else clause*** consists of the following parts:

- The keyword `else`, followed by
- a (block) statement

or

- The keyword `else`, followed by
- the keyword `if`, followed by
- an expression of type `boolean`, enclosed in parentheses, followed by
- a (block) statement, optionally followed by
- another `else` clause.

Execution of the `if` statement proceeds as follows:

First, the test expression of the `if` is executed. If its value is true, the (block) statement immediately following this test is executed. When this completes, execution continues after the end of the entire `if` statement, i.e., after the final `else` clause body (if any).

If the value of the first `if` test is false, execution continues at the first `else` clause. If this `else` clause does not have an if test, its body (block) is executed and then the `if` statement terminates. If the `else` clause does have an `if` test, execution proceeds as though this `if` were the first test of the statement, i.e., at the beginning of the preceding paragraph.

12.3 Limited Options: *switch*

An `if` statement is a very general conditional. Often, the decision of what action to take depends largely or entirely on the value of a particular expression. For example, in the calculator, the decision as to what action to take when a user presses a button can be made based on the particular button pressed. What we really want to do is to see which of a set of known values (all of the calculator's buttons) matches the particular value (the actual button pressed). This situation is sometimes called a ***dispatch on case***.

There is a special statement designed to handle just such a circumstance. In Java, this is a ***switch statement***. A `switch` statement matches a particular expression against a list of known values.

Before we look at the `switch` statement itself, we need to look briefly at the list of known values. In a Java `switch` statement, these values must be ***constant expressions***.

12.3.1 Constant Values

When we are choosing from among a fixed set of options, we can represent those options using symbolic constants. A ***symbolic constant*** is a name associated with a fixed value. For example, it would be lovely to write code that referred to the calculator's `PLUS_BUTTON`, `TIMES_BUTTON`, etc. But what values would we give these names? For that matter, what is the type of the calculator's *buttonID* ?

The answer is that it doesn't matter. At least, it doesn't matter as long as `PLUS_BUTTON` is distinct from `TIMES_BUTTON` and every other *buttonID* on the calculator. We don't want to add `PLUS_BUTTON` to `TIMES_BUTTON` and find out whether the value is greater or less than `EQUALS_BUTTON`, or to concatenate `PLUS_BUTTON` and `EQUALS_BUTTON`. But we do want to check whether

```
buttonID == PLUS_BUTTON
```

and the value of this expression ought to be (guaranteed to be) different from the value of

```
buttonID == TIMES_BUTTON
```

(unless the value of *buttonID* has changed). Contrast this with a constant such as **`Math.PI`**, whose value is at least as important as its name.

These symbolic constants, then, must obey a simple contract. A particular symbolic constant must have the same value at all times (so that **`EQUALS_BUTTON == EQUALS_BUTTON`**, always), and its value must be distinct from that of other symbolic constants in the same group (**`PLUS_BUTTON != EQUALS_BUTTON`**). These are the ONLY guaranteed properties, other than the declared type of these names.

12.3.1.1 Symbolic Constants

It is common, though not strictly speaking necessary, to declare symbolic constants in a class or interface rather than on a per instance basis. It makes sense for them to appear in an interface when they form part of the contract that two objects use to interact. For example, you might communicate with me by passing me one of a fixed set of messages — `MESSAGE_HELLO`, `MESSAGE_GOODBYE`, etc. — and the interface might declare these constants as a part of defining the messages that we both are expected to understand and use. This means that these symbolic constants are declared `static`.

It makes sense that a name such as this, which is part of a contract, might be declared `public`. This allows it to be used by any objects that need to interact with the symbolic constant's declaring object. Symbolic constants like this need not be public, but they often are. (Private symbolic constants would be used only for internal purposes. Package−level or protected symbolic constants might be used in a restricted way.)

In Java, a name is declared ***final*** to indicate that its value cannot change. This is one of the properties that we want our symbolic constants to have: unchanging value. A value declared final cannot be modified, so you need not worry that extra visibility will allow another object to modify a constant inappropriately.

It is common, though somewhat arbitrary, to use `ints` for these constants. There are some advantages to this practice, and it does simplify accounting. For example, by defining a set of these constants in sequence one place in your code, it is relatively easy to keep track of which values have been used or to add new values.

```
public static final int ...
                        PLUS_BUTTON = 10,
                        MINUS_BUTTON = 11,
                        TIMES_BUTTON = 12,
                        ...
```

Of course, you should *never* depend on the particular value represented by a symbolic constant (such as `EQUALS_BUTTON`), since adding a new symbolic name to the list might cause renumbering. The particular value associated with such a name is not important.

So ***symbolic constants*** are often `public static final ints`.

final

In Java, a name may be declared with the modifier ***final***. This means that the value of that name, once assigned, cannot be changed. Such a name is, in effect, ***constant***.

The most common use of this feature is in declaring ***final fields.*** These are object properties that represent constant values. Often, these fields are static as well as final, i.e., they belong to the class or interface object rather than to its instances. Static final fields are the only fields allowed in interfaces.

In addition to final fields, Java parameters and even local variables can be declared `final`. A ***final parameter*** is one whose value may not be changed during execution of the method, though its value may vary from one invocation of the method to the next. A ***final variable*** is one whose value is unchanged during its scope, i.e., until the end of the enclosing block.

[Footnote: final fields and parameters are not strictly speaking necessary unless you plan to use inner classes. They may, however allow additional efficiencies for the compiler or clarity for the reader of your code.]

Java methods may also be declared `final`. In this case, the method cannot be overridden in a subclass. Such methods can be inlined (i.e., made to execute with especially little overhead) by a sufficiently intelligent compiler.

Java classes declared `final` cannot be extended (or subclassed).

12.3.1.2 Using Constants

Properties such as the button identifiers are common to all instances of Calculators. In fact, they are reasonably understood as properties of the Calculator type rather than of any particular Calculator instance. They can (and should) be used in interactions between Calculator's implementors and its users. In general, symbolic names (and other constants) can be a part of the contract between users and implementors.

This means that it is often useful to declare these ***static final fields in an interface***, i.e., in the specification of the type and its interactions. In fact, static final fields are allowed in interfaces for precisely this reason. Thus, the definition of interfaces in Chapter 4, *Specifying Behavior: Interfaces*, is incomplete: interfaces can contain (only) abstract methods *and* static final data members.

For example, the Calculator's interface might declare the button identifiers described above:

```
public interface Calculator {
    public static final int PLUS_BUTTON = 10,
                            MINUS_BUTTON = 11,
                            TIMES_BUTTON = 12,
                            ...
                            EQUALS_BUTTON = 27;
}
```

Now any user of the Calculator interface can rely on these symbolic constants as a part of the Calculator contract. For example, the *isOperatorButton* predicate might be implemented as

```
public boolean isOperatorButton(int buttonID) {
    return (buttonID == Calculator.PLUS_BUTTON)
           || (buttonID == Calculator.MINUS_BUTTON)
           || (buttonID == Calculator.TIMES_BUTTON)
           || (buttonID == Calculator.DIVIDE_BUTTON);
}
```

[Footnote: Note the absence of any explicit conditional statement here. Using an `if` to decide which boolean to return would be redundant when we already have boolean values provided by `==` and by `||`. See the Style Sidebar on *Using Booleans* in Chapter 6, *Statements and Rules*.]

If we choose our numbering scheme carefully, the predicate *isDigitButton* could be implemented as

```
public boolean isDigitButton(int buttonID) {
    return (0 <= buttonID) && (buttonID < 10);
}
```

Of course, this is taking advantage of the idea that the digit buttons would be represented by the corresponding `int`s. This is a legitimate thing to do, but ought to be carefully documented, both in the method's documentation and in the declaration of the symbolic constants:

```
/**
 *  Symbolic constants representing calculator button IDs.
 *  The values 0..9 are reserved for the digit buttons,
 *  which do not have symbolic name equivalents.
 */
public static final int PLUS_BUTTON = 10,
                        MINUS_BUTTON = 11,
                        TIMES_BUTTON = 12,
                        ...
                        EQUALS_BUTTON = 27;
```

and

```
/**
 *  Assumes that the digit buttons 0..9 will be represented by
 *  the corresponding ints. These values should not be used for
 *  other buttonID constants.
 */
public boolean isDigitButton(int buttonID) {
    return (0 <= buttonID) && (buttonID < 10);
}
```

Style Sidebar

Use Named Constants

A ***constant*** is a name associated with a fixed value. Constants come in two flavors: ***constants that are used for their value***, and ***symbolic constants***, used solely for their names and uniqueness. *Calculator.PLUS_BUTTON* (whose value is meaningless) is a symbolic constant, while *Math.PI* (whose value is essential to its utility) is not. But constants — named values — are a good idea whether the value matters or not.

Introducing a numeric literal into your code is generally a bad idea. One exception is 0, which is often used to test for the absence of something or to start off a counting loop. Another exception is 1 when it is used to increment a counter. But almost all other numeric literals are hard to understand. In these cases, it is good style to introduce a name that explains what purpose the number serves.

Numbers that appear from nowhere, with no explanation and without an associated name, are sometimes called ***magic numbers*** (because they appear by magic). Like magic, it is difficult to know what kind of stability magic numbers afford. It is certainly harder to read and understand code that uses magic numbers.

In contrast, when you use a ***static final*** name, you give the reader of your code insight into what the value means. Contrast, for example, *EQUALS_BUTTON* versus `27`. You also decouple the actual value from its intended purpose. Code containing the name *EQUALS_BUTTON* would still work if *EQUALS_BUTTON* were initially assigned 28 instead of 27; it relies only on the facts that its value is unchanging and it is distinct from any other *buttonID*.

12.3.2 Syntax

We turn now to a ***switch*** statement. A `switch` statement begins by evaluating the expression whose value is to be compared against the fixed set of possibilities. This expression is evaluated exactly once, at the beginning of the execution of the `switch` statement. Then, each possibility is compared until a match is found. If a match is found, “body” statements are executed. A `switch` statement may also contain a default case that always matches. In these ways, a `switch` statement is similar to, but not the same as, a traditional conditional.

12.3.2.1 Basic Form

A simple `switch` statement looks like this:

```
switch (integralExpression) {
    case integralConstant:
         actionStatement;
         break;
    case anotherIntegralConstant:
         anotherActionStatement;
         break;
}
```

To execute it, first the *integralExpression* is evaluated. Then, it is compared to the first *integralConstant*. If it matches, the first *actionStatement* is executed. If *integralExpression* doesn't match the first *integralConstant*, it is compared to *anotherIntegralConstant* instead. The result is to execute the first *actionStatement* whose *integralConstant* matches, then jumps to the end of the `switch` statement.

For example, we might implement the calculator's *act* method like this:

```
switch (buttonID) {
    case Calculator.PLUS_BUTTON:
         this.handlePlus();
         break;
    // ...
    case Calculator.EQUALS_BUTTON:
         this.handleEquals();
         break;
}
```

The presence of the ***break statements*** as the last statement of each set of actions is extremely important. They are not required in a `switch` statement, but without them the behavior of the `switch` statement is quite different. See the *Switch Statement Sidebar* for details.

`Break` and `Continue` Statements

The `break` statement used here is actually more general that just its role in a `switch` statement.

A ***break statement*** is a general purpose statement that exits the innermost enclosing `switch`, `while`, `do`, or `for` block.

A variant form, the ***labeled break statement***, exits all enclosing blocks until a matching label is found. A labeled `break` does not exit a method, however. The labeled form of the `break` statement looks like this:

```
label:
    blockStatementText {
        bodyText
        break label;
        moreBodyText
    } endBlockStatementText
```

One or both of *blockStatementText* or *endBlockStatementText* may be present; for example, this block may be a `while` loop, in which case *blockStatementText* would be the code fragment **`while (expr)`** and there would be no *endBlockStatementText*.

[Footnote: The labeled block may be any statement containing a block, including a simple sequence statement. The body text may contain any statements, including — in the case of a labeled `break` — other blocks, so that a labeled `break` may exit multiple embedded blocks.]

This code is equivalent to:

```
try {
    blockStatementText {
        bodyText
        throw new LabelBreakException();
        moreBodyText>
    } endBlockStatementText
} catch (LabelBreakException e) {
}
```

[Footnote: Here, *LabelBreakException* is a unique exception type referring to this particular labeled `break` statement.]

That is, the labeled `break` statement causes execution to continue immediately after the end of the corresponding labeled block.

[Insert break pic, continue pic]

A similar statement, ***continue***, also exists in unlabeled and labeled forms.

An ***unlabeled continue statement*** terminates the particular body execution of the (`while`, `do`, or `for`) loop it is executing and returns to the (increment and) test expression.

The ***labeled continue statement*** works similarly, except that it continues at the test expression of an enclosing labeled `while`, `do`, or `for` loop. The labeled `continue` statement:

```
label:
    blockStatementText {
        bodyText
        continue label;
        moreBodyText
    } endBlockStatementText
```

is equivalent to

```
blockStatementText {
    try {
        bodyText
        throw new LabelContinueException();
        moreBodyText>
    } catch (LabelContinueException e) {
    }
} endBlockStatementText
```

12.3.2.2 The Default Case

In an `if` statement, if none of the test expressions evaluates to true, a final `else` clause without an `if` and test expression may be used as the default behavior of the statement. Such an `else` clause is always executed whenever it is reached.

In a `switch` statement, a similar effect can be achieved with a special case (without a comparison value) labeled ***default***:

```
switch (buttonID) {
    case Calculator.PLUS_BUTTON:
        this.handlePlus();
        break;
    // ...
    case Calculator.EQUALS_BUTTON :
        this.handleEquals();
        break;
    default:
        throw new NoSuchButtonException(buttonID);
}
```

If no preceding case matches the value of the test expression, the `default` will always match. It is therefore usual to make the `default` the final case test of the `switch` statement. (No case after the `default` will be tested.) When the `default` clause is the last statement of your `switch`, it is not strictly speaking necessary to end it with a `break` statement, though it is not a bad idea to leave it in anyway. The final `break` statement is omitted in this example because it would never be reached after the `throw`. (Any instruction follower executing the `throw` would exit the `switch` statement at that point.)

It is often a good idea to ***include a default case***, even if you believe that it is unreachable. You would be amazed at how often "impossible" circumstances arise in programs, usually because an implicit assumption is poorly documented or because a modification made to one part of the code has an unexpected effect on another.

[Insert switch/default pic]

12.3.2.3 Variations

It is possible to write a `switch` statement without using `break`s. In this case, when a case matches, not only its following statements but all statements within the `switch` and up to a `break` or the end of the `switch` statement will be executed. This can be useful when the action for one case is a subset of the action for a second case. **Beware:** of accidentally omitted `break` statements in a `switch` . Because omitting the `break` is sometimes what you want, it is legal Java and the compiler will not complain. Omitting a `break` statement will cause the statements of the following case(s) to be executed as well.

If two (or more) cases have the same behavior, you can write their cases consecutively and the same statements will be executed for both. This is, in effect, giving the first case no statements (and no `break`) and letting execution “drop through” to the statements for the second case. For example:

```
switch (buttonID) {
    case Calculator.PLUS_BUTTON:
    case Calculator.MINUS_BUTTON:
    case Calculator.TIMES_BUTTON:
    case Calculator.DIVIDED_BY_BUTTON:
        this.handleOperator(buttonID);
        break;
    // ....
}
```

In this `switch` statement, the same action would be taken for each of the four operator types. The *buttonID* pressed is passed along to the operator handler to allow it to figure out which operator is needed.

12.3.2.4 *Switch* Statement Pros and Cons

A `switch` statement is very useful when dispatch is based on the value of an expression and the value is drawn from a known set of choices. The `switch` expression must be of an integral type and the comparison case values must be constants (i.e., literals or final names) rather than other variable names. **When a `switch` statement is used, the `switch` expression is evaluated only once.**

A `switch` statement cannot be used when the dispatch expression is of an object type or when it is a floating point number. It also cannot be used with a boolean, but since the boolean expression has only two possible values, an `if` statement with a single alternative makes at least as much sense in that case.

The requirement that a `switch` expression must be of integral type is one reason why `static final ints` are often used as symbolic constants. `int` is a convenient integral type and symbolic constants are naturally compatible with `switch` statements.

A `switch` statement cannot be used when the comparison values are variable or drawn from a non−fixed set. That is, if the dispatch expression must be compared against other things whose values may change, the `switch` statement is not appropriate. For example, you wouldn't want to use a `switch` statement to compare a number against the current ages of the employees of your company, because these are changing values.

The `switch` statement is also not appropriate for expressions that may take on any of a large range of values. (“Large” is subjective, but if you wouldn't want to write out all of the cases, that's a good indication that you don't want a `switch` statement.) For example, you wouldn't want to do a dispatch on a the title of a returned library book, testing it against every book name in the card catalog, even if you represented names as symbolic constants rather than as Strings.

[Footnote: Of course, if you represented the names as Strings, you couldn't use a `switch` statement because String is an object type.]

Switch Statement Syntax

A ***switch statement*** contains a ***test expression*** and at least one ***case clause***. After that, the `switch` statement may contain any number of case clauses or statements in any order:

```
switch (integralExpression) {
    caseClause
    caseClauses or statements
}
```

The ***integralExpression*** is any expression whose type is an integral type: `byte`, `short`, `int`, `long`, or `char`.

A ***caseClause*** may be either

```
case constantExpression:
```

or

```
default:
```

If the *`caseClause`* contains a *`constantExpression`*, this must be an expression of an integral type whose value is known at compile time. Such an expression is typically either a literal or a name declared `final`, although it may also be an expression combining other constant expressions (e.g., the product of a literal and a name declared `final`).

Note that each *`caseClause`* must end with a colon.

Typically, the actual syntax of a `switch` statement is:

```
switch (integralExpression) {
    case constantExpression:
    case constantExpression:
    ...
    case constantExpression:
        statements
        break;

    ...

    default:
        statements
        break;  // optional
}
```

12.4 Arrays

Sometimes, what we really want to do when dispatching is to translate from one representation to another. For example, in constructing a Calculator, we might want to move from the symbolic constants used to identify buttons above to the actual labels appearing on those buttons. We might even want to move between the labels on buttons and the buttons themselves. If our collection of objects is indexed using an integral type — either because it is naturally indexed or because we have used `int`s as symbolic constants — we can often accomplish this conveniently using *arrays*.

[Mailbox pic]

12.4.1 What is an Array?

An ***array*** is an integrally indexed grouping of dials or labels. You can think of it sort of like a wall full of numbered mailboxes. In identifying a mailbox, you need to use both a name corresponding to the whole group ("the mailboxes in the lobby") and an index specifying which one ("mailbox 37"). Similarly, an array itself is a thing that can be

named — like the group of mailboxes — and it has members — individual mailboxes — named using both the array name *and* the index, in combination. For example, my own particular individual mailbox might be named by *lobbyMailboxes*[37].

An array has an associated type that specifies what kind of thing the individual names within the array can be used to refer to. This type is sometimes called the ***base type of the array***. For example, you can have an array of `chars` or an array of Strings or an array of Buttons. The individual names within the array are all of the same type, say `char` or `String` or `Button`.

That is, an array is a ***collection*** of nearly−identical names, distinguished only by an `int` index. An array of dial−type—for example, an array of `chars`—really is almost like a set of mailboxes, each of which is an individual dial−name. To identify a particular dial, you give its mailbox number. For example, you can look and see what (`char`) is in mailbox 32 or put an appropriately typed thing (`char`) in mailbox 17. Label−type arrays work similarly, though it's hard to find an analogously appropriate analogy. (A set of dog−tags or post−it notes is along the right lines, but it is harder to visualize these as neatly lined up and numbered.) A label−type array — such as an array of Buttons − is an indexed collection of labels suitable for affixing on things of the appropriate type — such as Buttons. The names affixed on individual Buttons are names like *myButtons*[8];, the ninth button in my array.

[Footnote: Yes, that's right, *myButtons*[8], the ninth button. Array elements, like the characters in Strings, are numbered starting from 0.]

[Label array pic]

12.4.1.1 Array Declaration

An ***array type*** is written just like the type it is intended to hold, followed by square braces. For example, the type of an array of `chars` is `char`[] and the type of an array of Buttons is `Button[ ]`. Note that, like `char` and Button, `char`[] and `Button[ ]` denote types, not actual Things. So, for example,

```
char[] initials;
```

makes the name *initials* suitable for sticking on things of type `char[]`; it doesn't *create* anything of type `char[]` or otherwise affix *initials* to some Thing. Similarly,

```
Button[] pushButtons;
```

creates a label, *pushButtons* , suitable for attaching to a `Button[]`, and nothing more. Note that both *initials* and *pushButtons* are *label* names, not *dial* names. The names of array types are always label types, although a particular array may itself be suitable either for holding dial (e.g., `char`) or label (e.g. Button) types.

[Array declaration and construction pic]

12.4.1.2 Array Construction

To actually create a `char[]` or `Button[]`,

[Footnote: Pronounced “Button array” or “array of Buttons”.]

you need an ***array construction*** expression. This looks a bit like a class instantiation expression, but it is actually not quite the same. An ***array construction expression*** consists of the keyword `new` followed by the array type with an array size inside the square braces. For example,

```
new char[26]
```

is an expression that creates 26 `char`–sized mailboxes, numbered 0 through 25. Similarly,

```
new Button[518]
```

is an expression whose value is a brand new array of 518 Button–sized labels. Note that arrays are indexed starting at 0, so the last index of a member of this array will be 517, one less than the number supplied to the array construction expression.

The expression

```
pushButtons = new Button[numButtons]
```

makes the name *pushButtons* refer to a new array of `Button`–sized labels. How many? That depends on the value of *numButtons* at the time that this statement is executed.

The statement

```
String[] buttonLabels = new String[16];
```

combines all of these forms, creating a name (*buttonLabels*) suitable for labeling an array of Strings (`String`[]), constructing a 16–`String` array, and then attaching the name *buttonLabels* to that array. Note that the text **`String[]`** appears *twice* in this definition, once as the type and once (with an integral argument between the brackets) in the array construction expression. 12.4.1.3 Array Elements

To access a particular member of the array, you need an expression that refers to the array (such as its name), followed by the index of the particular member inside square braces. For example,

```
buttonLabels[2]
```

is an expression of type String that refers to the element at index 2 of the String array named by *buttonLabels*. Recall that, since the indices of *buttonLabels* run from 0 to 15, **`buttonLabels[2]`** is the *third* element of the array.

This expression behaves very much as though it were a name expression. Like a name, an ***array element expression of label type*** may be stuck on something, or may be `null`. An ***array element of dial type*** (e.g., **`initials[6]`**) behaves like a dial name.

You can use these ***array member expressions*** in any place you could use a name of the same type. So, for example, you can say any of the following things:

```
buttonLabels[2] = "Hi there";
String firstString = buttonLabels[0];
buttonLabels[7] = buttonLabels[6] + buttonLabels[5];
Console.println(buttonLabels[Calculator.PLUS_BUTTON]);
if (buttonLabels[currentIndex] == null) ...
```

(assuming of course that **`Calculator.PLUS_BUTTON`** and **`currentIndex`** are both integral–type names).

Array Syntax

Array Type

An ***array*** is a label name whose type is any Java type followed by []. The array is an array *of that type*. Admissible types include dial (primitive) types, label (object) types, and other array types. An array is declared like any other Java name, but using an array type. For example, if *baseType* is any Java type, then the following declaration creates a label, *arrayName*, suitable for affixing on an *array of baseType*:

```
baseType[] arrayName;
```

Array Initialization

By default, an array name's value is `null`. An array name may be defined at declaration time using an ***array literal***. This consists of a sequence of comma-separated constant expressions enclosed in braces:

```
baseType[] arrayName = { const0, const1, ...  constN };
```

Array Construction

Unless an array initialization expression is used in the declaration, an array must be constructed explicitly using the array construction expression

```
new baseType[size]
```

Here, `baseType` is the base type of the array (i.e., this expression constructs an *array of* `baseType`) and `size` is any non-negative integral expression.

Array Access

The expression `arrayName[index]` behaves as a “regular” Java name. Its type is the array's base type. The expression inside the square brackets is called the ***index*** of the ***array access expression***. The first index of an array is 0. The ***length of an array*** *arrayName* is given by the expression **`arrayName.length`**. Thus, arrays are numbered from 0 to **`arrayName.length - 1`**. Attempting to access an array with an index outside this range throws an ***ArrayOutOfBoundsException***.

12.4.2 Manipulating Arrays

The particular names associated with individual members of an array behave like

ordinary (dial or label) names. What is unusual about them is how you write the name — ***arrayName[index]*** — rather than how they actually behave.

You can find out how many elements are in a particular array with the expression ***arrayName*.length**. Note that there are no parentheses after the word ***length*** in this expression. Technically, this is not either a field access or a method invocation expression, although it looks like one and behaves like the other.

Note also that the value of the expression ***arrayName*.length** is *not* the index of the last element of the array. It is in fact one more than the final index of the array, because the array's indices start at 0. Attempting to access an array element with a name smaller than 0 or greater than or equal to its length is an error. In this case, Java will throw an ***ArrayOutOfBoundsException***.

[Picture of changing array references]

Once you construct an array, the number of elements in that array does not change. However, this immutable value is the number of elements in the array itself, not the number of elements associated with the array's name. If the name is used to refer to a different array later, it may have a different set of legal indices. For example:

```
char[] firstInitials = new char[10];
   // firstInitials[3] would be legal,
   //  but firstInitials[12] would not.

firstInitials[5] = 'f';
firstInitials[5] = 'g';
   // changes the value associated with a particular mailbox

firstInitials = new char[2];
   // changes the whole set of mailboxes
   // now firstInitials[3] isn't legal either!
```

12.4.2.1 *Stepping Through an Array* Using a *for* Statement

One common use of arrays is as a way to ***step through a collection*** of objects. If you are going to work your way through the collection, one by one, it is common to do so using a counter and a loop.

We can write this with a `while` loop:

```
int index = 0;

while (index < arrayName.length) {
    // do something
    index++;
}
```

Note that *index* can't be initialized inside the `while` statement or it wouldn't be bound in the test expression. Local (variable) names have scope only from their declarations until the end of their enclosing blocks.

This is so common, there's a special statement for it. The `while` statement above can be replaced by:

```
for (int index = 0; index < arrayName.length; index++) {
    // do something
}
```

Note that the `for` loop also includes the declaration of *index*, but that *index* only has scope inside the `for` loop. It is as though *index*'s definition plus the `while` loop were enclosed in a block.

For additional detail on `for` statements, refer to the sidebar.

For Statement Syntax

The syntax

```
for (initStatement; testExpression; incrementStatement) {
    body
}
```

is the same as

```
{
    initStatement;

    while (testExpression) {
        body
        incrementStatement;
    }
}
```

The expression *testExpression* is any single boolean expression. It falls within the scope of any declarations made in *initStatement*.

Both *initStatement* and *incrementStatement* are actually allowed to be multiple statements separated by commas, e.g.:

```
i = i + 1, j = j + i
```

Note that *initStatement*, *testExpression*, and *incrementStatement* are separated by *semicolons*, but that individual statements within *initStatement* and *incrementStatement* are separated by *commas*. There is no semicolon at the end of *incrementStatement*.

12.4.3 *Using Arrays for Dispatch*

In addition to their use as ***collection objects***, arrays can be used as a ***mechanism for dispatch***. This is because the value that you have been asked to respond to — the value upon which you are dispatching — can, under some circumstances, be stored in an integral−type variable and used as an index into an array. As the value of that variable changes, and hence as the index of the array access changes, you get different behavior.

The rest of this subsection gives an example of using an ***array for dispatch***. In this example, we are not going to use an array to do the calculator's central dispatch job. Instead, we will consider the problem of constructing actual GUI Button objects that will appear on the screen.

There should be one Button corresponding to each of the symbolic constants described above. Each of these Buttons will need an appropriate label, to be passed into the Button constructor. We might create a method,

```
String getLabel(int buttonID)
```

for this purpose.

We could use our *getLabel* to say

```
new Button(this.getLabel(buttonID))
```

or even

```
gui.add(new Button(this.getLabel(buttonID)))
```

Such a *getLabel* method, which could translate from *buttonID*'s to labels, would also be useful for generating Strings suitable for printing to the Console, e.g., for debugging purposes.

One way to implement this method would be with an `if` statement. In this case, the body of the method might say:

```
if (buttonID == Calculator.PLUS_BUTTON) {
    return "+";
} else if (buttonID == Calculator.MINUS_BUTTON) {
    return "-";
} else if (buttonID == Calculator.TIMES_BUTTON) {
    // and so on....
}
```

Of course, this would get rather verbose rather quickly.

Because we are really doing a dispatch on the value of *buttonID*, and because we've cleverly chosen to implement these symbolic constants as `ints`, we could opt instead to use a `switch` statement:

```
switch (buttonID) {
    case Calculator.PLUS_BUTTON:
        return "+";
    case Calculator.MINUS_BUTTON:
        return "-";
    // and so on....
}
```

This may be somewhat shorter, but not much. It does have the advantage of making the dispatch on *buttonID* more explicit. But we can do still better. **Question:** In the immediately preceding `switch` statement, why are there no `break` statements?

If we create an array containing the button labels, in order, corresponding to the *buttonID* symbolic constants, then we can use the *buttonID* to select the label:

```
String[] buttonLabels = { "0", "1", "2", "3", "4",
                          "5", "6", "7", "8", "9",
                          "+", "-", "*", "/",
                          // and so on ... up to
                          "="
};
```

In this case, the entire body of our *getLabel* method might say simply

```
return this.buttonLabels[buttonID];
```

This example is relatively simple, but in general arrays can be used whenever there is an ***association*** from an ***index set*** (such as the *buttonID*s) to other values. The idea is that the index pulls out the correct information for that particular value. This is a very simple form of a very powerful idea, which we shall revisit when we discuss object dispatch in Chapter 14, *Intelligent Objects and Implicit Dispatch*.

12.5 When to Use Which Construct

Arrays are in many ways the most limited of the dispatch mechanisms. They work well when the action is uniform up to some integrally indexed decisions, e.g., some integrally indexed variables need to be supplied. Setting up the array appropriately allows for very concise code. This is not always possible, though, either because there isn't an obvious index set, because the index set is not integral, because it is not possible to set up the necessary association, or because the needed responses are nonuniform.

Switch statements also rely on integrally indexed decisions on a single expression, but they are otherwise quite general in the action(s) that can take place. They are useful any time the decision is made by testing the expression against a pre-known fixed set of constants. In other words, a `switch` statement can be used whenever an array is appropriate, though it may be more verbose. A `switch` statement can also be used in cases of nonuniform response, where an array would not be appropriate.

`Ifs` are very general. You can do anything with them. You should use them when none of the other mechanisms are appropriate.

In a subsequent chapter, we will see an additional dispatch mechanism, ***object dispatch***, that resembles the implicit nature of array-based dispatch, but without many of its restrictions.

Chapter Summary

- ***Dispatch*** is the process of deciding what action needs to be taken based on one's input. It is essentially a middle management function.

- ***Conditional statements*** are used when a piece of code should be executed under some but not all circumstances.

  - An ***if statement*** may consist only of a single boolean test expression and a body. This body is executed only if the test expression's value is `true`.

  - An ***if statement*** may optionally have an `else` clause with a body that is executed only when the `if`'s test expression has the value `false`.

  - The `else` clause of an `if` statement may itself be an `if` statement. In this case, it is preferable to use ***cascaded*** rather than ***embedded*** `if`s.

  - Each test expression is evaluated independently as it is reached.

- Numbers generally should not appear in code. Instead, use ***symbolic constants*** with descriptive names.

- A ***switch statement*** is used when different actions must be taken depending on the value of a single expression.

  - This expression is evaluated only once. Its type must be integral.

  - In a `switch` expression, the value is compared against different cases, which must be constants. Once a case matches, the statements of the `switch` body are executed until either a `break` or the end of the `switch` body is reached.

  - Switch has a specialized case, ***default***, which always matches.

- An ***array*** is a uniformly typed ***collection*** of names.

  - The type of the array member names is the array's base type. The array member names may be either dial names or label names, depending on the base type.

  - The type of the array is "array of *base type*." The array name is a label name.

- The names of array members are written using the array name followed by an integral index enclosed in square brackets.

- The indices of an array run from 0 to ***`arrayName`*****`.length - 1`**.

- Like an object, an array must be explicitly created using `new`.

Exercises

1. Answer the Question that appears at the end of Section 12.2.4, *Many Alternatives*.

2. Convert the following to a `for` loop:

```
int sum = 0;
int i = 1;
while (i < this.MAXIMUM) {
    sum = sum + i;
    i = i + 2;
}
```

3. Write a method that takes an array of `ints` and returns the sum of these `ints`.

4. Consider a certain class, say, the Foo class.

    a. Write the statement(s) that would declare a field named *transformers* suitable for an array of StringTransformers.

    b. Write the statement(s) that would define *transformers* to be an array of 40 StringTransformers.

    c. Write the statement(s) that would set each element of *transformers* to be a Capitalizer. Write your statement(s) in such a way that you do NOT need to assume that there are 40 elements in the array.

    d. Assume that *transformers* has been filled with StringTransformers of various kinds. (So, contrary to the previous parts of this question, the StringTransformers in the array need NOT all be Capitalizers, and there need NOT be 40 of them.) Write a method *produceAllTransformations*, that takes in a String and returns an array of Strings, where:

        ◊ The first element of the returned array corresponds to the transformation of the argument String by the first transformer in the *transformers* array.

        ◊ The second element of the returned array corresponds to the transformation of the argument String by the second transformer in the *transformers* array.

        ◊ And so on.

    Recall that a StringTransformer implements an interface which requires that it has a *transform* method that takes a String and returns a (transformed) String.

5. Consider the following code, excerpted from the definition of class EmotionalSpeaker.

```
public String transformEmotionally(Type emotion, String what) {
    switch (emotion) {
        case EmotionalSpeaker.HAPPY:  return sayHappily(what);
        case EmotionalSpeaker.SAD:    return saySadly(what);
        case EmotionalSpeaker.ANGRY:  return sayAngrily(what);
    }
}
```

where, e.g.,

```
private String sayHappily(String what) {
    return "I'm so happy that ";
}
```

(You may assume similar definitions for the other emotions, with appropriate modifications.)

Define the symbolic constants *HAPPY*, *SAD*, and *ANGRY*, and provide a type for *emotion*.

6. In the previous exercise, the `switch` statement contains no `breaks`. What gets printed when we invoke:

```
transformEmotionally(EmotionalSpeaker.SAD, "I am here.")
```

7. Continuing the previous two exercises: Using an array, modify the body of *transformEmotionally* so that it is a single, short statement that easily fits on a single line. Assume that the array was elsewhere defined appropriately.

Chapter 13

Encapsulation

Chapter Overview

- How do I package up implementation details so that a user doesn't have to worry about them?

- How do I make my code easier to read, understand, modify, and maintain?

Good design separates use from implementation. Java provides many mechanisms for accomplishing this. In this chapter, we review a variety of mechanisms that allow this sort of separation.

Procedural abstraction is the idea that each method should have a coherent conceptual description that separates its implementation from its users. You can ***encapsulate*** behavior in methods that are internal to an object or methods that are widely usable. Methods should not be too complex or too long. Procedural abstraction makes your code easier to read, understand, modify, and reuse.

Packages allow a large program to be subdivided into groups of related classes and instances. Packages separate the names of classes, so that more than one class in a program may have a given name as long as they occur in different packages. In addition to their role in naming, packages have a role as ***visibility protectors***. Packages provide visibility levels intermediate between public and private. Packages can also be combined with inheritance or with interfaces to provide additional encapsulation and separation of use from implementation.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

Inner classes are a mechanism that allows one class to be encapsulated inside another. Perversely, you can also use an inner class to protect its containing class or instance. Inner classes have privileged access to the state of their containers, so an inner class can provide access without exposing the object as a whole.

Objectives of this Chapter

1. To understand how information−hiding benefits both implementor and user.
2. To learn how to use procedural abstraction to break your methods into manageable pieces.
3. To be able to hide information from other classes using visibility modifiers, packages, and types.
4. To recognize inner classes.

13.1 Design, Abstraction, and Encapsulation

This chapter is about how information can be hidden inside an entity. There are many different ways that this can be done. Each of these is about keeping some details hidden, so that a user can rely on a commitment, or contract, without having to know how that contract is implemented. There are numerous benefits from such information hiding.

First, it makes it possible to use something without having to know in detail how it works. We do this all the time with everyday objects. Imagine if you had to understand how a transistor works to use your computer, or how a spark plug works to use your car, or how atoms work to use a lever.

Second, information−hiding gives some flexibility to the implementor. If the user is not relying on the details of your implementation, you can modify your implementation without disturbing the user. For example, you can upgrade your implementation if you find a better way to accomplish your task. You can also substitute in different implementations on different occasions, as they may become appropriate.

Finally, hiding information is liberating for the user, who does not expect nor make great commitment to particulars of the implementation. The name for this idea — of using more general properties to stand in for detailed implementation — is ***abstraction***. To facilitate abstraction, it is often convenient to package up the implementation details into a single unit. This packaging−up is called ***encapsulation***.

13.2 Procedural Abstraction

Procedural abstraction is a particular mechanism for separating use from implementation. It is tied to the idea that each particular method performs a well−specified function. In some cases, a method may calculate the answer to a particular question. In others, it may ensure the maintenance of a certain condition or perform a certain service. In all cases, each method should be accompanied by a succinct and intuitive description of what it does.

[Footnote: It is not, however, essential that a method have a succinct description of *how it does* what it does. How it accomplishes its task is an implementation detail.]

A method whose function is not succinctly describable is probably not a good method. Conversely, almost every succinctly describable function should be a separate method, albeit perhaps a private or final one.

This idea, that each conceptual unit of behavior should be wrapped up in a procedure, is called ***procedural abstraction***. In thinking about how to design your object behaviors, you should consider which chunks of behavior — whether externally visible or for internal use only — make sense as separate pieces of behavior. You may choose to encapsulate a piece of behavior for any or all of the following reasons:

- It's a big, ugly function and you want to hide the “how it works” details from code that might use it. Giving it a name allows the user to ignore how it's done.

- It's a common thing to do, and you don't want to have to replicate the code in several places. Giving it a name allows multiple users to rely on the same (common) implementation.

- It's conceptually a separate “task”, and you want to be able to give it a name.

Note also that the behavior of a method may vary slightly from invocation to invocation, since the parameters can influence what you the code actually does.

13.2.1 The Description Rule of Thumb

Each method in your program should have a well−defined purpose, and each well−defined purpose in your program should have its own method. You should be able to succinctly state what each method in your program does. If you cannot, your methods are either too large (i.e., should be broken into separable conceptual units) or too small (i.e., should be combined so that each performs a “complete” task.

Note that having a succinct description of what a method does is quite different from being to state succinctly how it accomplishes this. It is unfortunately all too common that a method's implementation is obscure. It is important that the user understand when, why,

and under what circumstances your method should be used, i.e., what it does. You provide a method precisely so that the user will not have to understand how your method works.

For example, it is common to test complex conditions using a single predicate. One such instance might be the Calculator's `isDigitButton()` method, which determines whether a particular Calculator button represents the digits 0 through 9 (or instead is, e.g., an arithmetic operator). The logic behind `isDigitButton()` might be somewhat obscure. However, it is easy to succinctly state what the method determines and, therefore, when and why you might use it. This use of predicates as abstractions make code for easier to read, decompose, and understand.

The importance of succinct summarizability does not mean that there is exactly one method per description. For example, one succinctly summarizable method may in turn rely on many other succinctly summarizable methods. This is the "packaging up substeps" idea from Chapter 1: making a sandwich may be described in terms of spreading the peanut butter, spreading the jelly, closing and cutting the sandwich. Each substep may itself be a method. When the substeps are not likely to be useful for anything except the larger method of which they are a part, these methods should be private to their defining class.

It may also be the case that multiple methods each implement the same well−defined purpose. For example, multiple similar methods may operate on different kinds of arguments. A method that draws a rectangle may be able to take a java.awt.Rectangle, two java.awt.Points, or four ints as arguments. Each of these methods will have a different signature. They may, however, rely on a common (shared) method to actually perform much of the work, sharing as much code as possible. (See the repetition rule of thumb, below.)

Or it may be the case that multiple distinct object types each have similar methods performing similarly summarized functions. In this case, it may make sense to have a common interface implemented by each of these classes, documenting their common purpose. Occasionally it even makes sense to split off the method into its own class, turning instances of the new class into components of the old. (See the discussion of using contained objects in the chapter on Object Oriented Design.)

When a single method does too many things, it can be difficult to decide whether you want to invoke it. It can be awkward to figure out what it is really doing. And the interdependencies among subtasks can make your code hard to maintain, especially if the assumptions that caused you to bundle these pieces together no longer hold.

Succinct summarizability makes your code immensely easier to read. By choosing descriptive names, you can often make your code read like the English description of what it does. This makes it easier to read, understand, modify, and maintain your code.

13.2.2 The Length Rule of Thumb

A single method should ideally fit on a single page (or screen). Often a method will only be a few lines long. If you find yourself writing longer methods, you should work on figuring out how to break them up into separable substeps. The description rule of thumb is handy here.

When a method's implementation takes up too much space, it is difficult to read, understand, or modify. It can be hard to hold the whole method in your head. It can be overwhelming to try to figure out what it is actually doing.

Appropriate method length is a matter of some individual judgment. Some people don't like to write methods longer than a half−page. Others regularly write much longer methods. As you become a more skilled programmer, you will become accustomed to keeping track of larger and more complex programs. But more complex programs do *not* mean longer methods. It will always be the case that brevity of individual units — such as methods — makes the overall flow easier to understand. Mnemonic names (describing what the method accomplishes) and programs that read like English descriptions of their behavior (through the use of well−chosen names) make your code more comprehensible to subsequent readers.

How do you know when to break code into pieces? If you discover that you have written a method that does not fit on a single page, you should write an outline for how the code works. Each of the major steps of this outline should be turned into a method. The original code should be rewritten in terms of these methods. The major steps should now be shorter methods. If these are still too long, repeat this process until each piece of code has a succinct description and occupies no more than two pages of code.

Note: Do not worry about inefficiency created by having too many small methods. First, intelligible code is so much easier to read and maintain, and code carefully optimized for efficiency so much more difficult to work with, that it rarely pays to do this sort of optimization until you are a skilled programmer. Further, a good compiler should be able to optimize. For example, if you make a method private or final, the compiler can in−line it.

13.2.3 The Repetition Rule of Thumb

Any time that the same code appears in two different places, you should consider capturing this common patterns of usage in a single method. When this happens, it is often because there is an idea expressed by this code. It is useful to give this idea a name, and to encapsulate or abstract it for reuse. Even if there are minor differences in the code as it appears, you may be able to abstract to a common method by supplying the distinguished information as arguments to the method. Each of the original pieces of code should be rewritten to use the common method.

Methods created by abstracting two or more pieces of code within the same class are often declared private. This is appropriate whenever the common behavior is local to the particular object and not something you want to make generally available. At other times, though, the common code is a useful and nameable function on its own. Though you may discover the commonality by replicating code, the existence of a separate method to replace this redundancy can be turned into an opportunity to export this functionality if it should make sense to do so.

Combining redundant code is also important in the case of constructors. Constructors can share code by having one invoke another — using the special `this()` construct — or by using a call to one or more (private) helper methods. A common programming mistake is to modify only one constructor when in reality the same change must be made to every constructor. Having the bulk of the work of the constructor done by a common method (or shared by using `this()`–constructors) eliminates this error.

Sharing redundant code shortens your program, making it easier to read, understand, modify, and maintain. It also helps to isolate a single point where each piece of behavior is performed. This single point can be understood, modified, and debugged once rather than each time it (redundantly) appears.

13.2.4 Example

In the example immediately below, we will modify code based on redundancy, i.e., the repetition rule of thumb. The result will also make our code more succinct and easier to read. The newly created method will be succinctly summarizable and a legitimately separable subtask.

Consider a bank account, which might have a method that allows the account's owner to obtain balance information: int getBalance(Signatory who) throws InvalidAccessException { if (! who == this.owner) { throw new InvalidAccessException(who, this) } // else return this.balance; }

It might also have a withdraw method that allows the owner to remove amount from the account, returning that amount as cash: public Instrument withdraw(int amount, Signatory who) throws InvalidAccessException { if (! who == this.owner) { throw new InvalidAccessException(who, this) } // else this.balance = this.balance − amount; return new Cash(amount); }

We could abstract the common pattern here, which is the verification of a signatory's right to access this account: private void verifyAccess(Signatory who) throws InvalidAccessException { if (! who == this.owner) { throw new InvalidAccessException(who, this) } }

Now, we can rewrite getBalance and withdraw: int getBalance(Signatory who) throws InvalidAccessException { this.verifyAccess(who); return this.balance; } public

Instrument withdraw(int amount, Signatory who) throws InvalidAccessException {
this.verifyAccess(who); this.balance = this.balance − amount; return new Cash(amount);
}

Much simpler, much more succinct, and in addition if we later need to modify the access verification routine, there is only a single place — `verifyAccess()` — where changes will need to be made.

Style Sidebar

Procedural Abstraction

- Use procedural abstraction when a method call would make your code (at least one of)
    - shorter, or
    - easier to understand.
- Your method should be concisely describable as "single function", though the function may itself have many pieces.
- Use parameters to account for variation from one invocation to the next.
- Return a value when the target of an assignment varies; leave the actual assignment out of the method body.
- Share code where possible. This is especially true among constructors, where one constructor can call another using this().
- Make internal helper procedures private. Make generally useful common functionality public (or protected).

13.2.5 Benefits of Abstraction

Abstracting procedures — creating short, succinctly describable, non−redundant methods — has many benefits. Even in the simple example of the preceding section, we can see many of these.

Procedural abstraction makes it easier to read your code, especially if methods have names corresponding to their succinct descriptions and the flow of code reads like the logic of the English description. Compare the before−and−after withdrawal methods of the bank account in the previous section.

Greater readability makes it easier to understand and figure out how to modify and maintain code. Separating functionality into bite−sized pieces also creates many opportunities to modify individual methods. Sharing these methods also centralizes the locations needing modification. For example, we could add a digital signature check to the verification procedure of the bank account by modifying only verifyAccess, not the bodies of getBalance or withdraw.

In contrast, long methods with complicated logic can be particularly hard to modify, either because their interconnected logic can be so difficult to understand or because it can be hard to find the right place to make the change.

As the needs of your code change, you will also find it easier to rearrange and reconfigure what your code does if the logical pieces of the code are separated. For example, we might add a wireTransfer method to the bank account. In doing so, we can reuse the verifyAccess method.

Of course, smaller methods make for bite−sized debugging tasks. It is much easier to see how to debug access verification in the newer bank account than in the version where each account interaction has its own verification code and where verification is intimately intertwined with each transaction. And if we need to modify the verification procedure — to give diagnostic information, to step through the method, or to fix it — there is a central place to make these changes.

Procedural abstraction also makes it easier to change behavior by substituting a new version of a single method. If a method is not private, it can be overridden by a subclass, specializing or modifying the way in which it is carried out without changing its succinct specification. We could, for example, have a more secure kind of bank account using the digital signature verification method alluded to above.

Many of the advantages of procedural abstraction are also provided by good object design. A method signature is a reasonable abstraction of the behavior of an individual method. An interface plays a similar role for an entire object, packaging up (encapsulating) the behavioral contract of an object so that its particular implementation may vary. Interfaces also make it easier to see how a single abstraction can have many coexisting implementations.

13.3 Protecting Internal Structure

Procedural abstraction is an important way to separate use from implementation and a significant part of good program design. Procedural abstraction is not the only kind of abstraction that you need in a program, though. Often, other techniques are used, either alone or with procedural abstraction, to hide implementation details. For example, if you use procedural abstraction to create local helper methods, you generally will not want these helper methods to be available for other objects to use.

In this section, we will look at several ways to protect internal structure — such as helper methods — from use by others. These techniques protect implementation by making parts of the inner structure of an object inaccessible from outside that object or that group of interrelated objects. This packaging of internal structure is another kind of encapsulation. This section discusses some Java−specific ways to encapsulate functionality. Many programming languages offer similar mechanisms.

13.3.1 *private*

One of the most straightforward ways to protect internal structure — such as fields or helper methods — is to declare them private. We have seen in the section above how private methods can be used for procedural abstraction — to break up a long procedure, to capture common patterns, etc. — without exposing these functions to other objects. A method (or other member) declared private can only be called from within the class.

Beware: This is not the same thing as saying that only an object can call its own private methods. An object can call the private methods of any other instance of the same class.

Private is extremely effective at protecting methods and other members from being used by other objects. However, a member declared private cannot be accessed from code within a subclass. This means that if you modify code in a subclass that relies on a private helper method in the superclass, you will have to recreate that private helper method.

13.3.2 Packages

An alternative to the absolute protection of private is the use of packages. A package is a collection of associated classes and interfaces. You can define your own packages. Libraries — such as the Java source code or the cs101 distribution — generally define packages of their own. The association among classes and interfaces in a package can be as loose or as tight as you wish to make it.

Sometimes the association among objects is merely by convenience: many kinds of objects deal with the same kind of thing. Most of the cs101 packages are of this sort. Often, it makes sense to define a set of interrelated classes and interfaces in a single package and to provide only a few entry points into the package, i.e., a few things that are usable from outside the package. These packages represent associations by shared interconnectedness. Most of the interlude code is of this sort. Java defines a large number of packages, some of each kind.

In the bank account, we might well choose to define the interface `Instrument` (representing cash and checks, among other things) and classes `BankAccount`, `CheckingAccount`, `Cash`, etc. in a single package, say `finance`.

Packages play two roles in Java. The first concerns names and nicknames. Packages determine the proper names of Java classes and interfaces. The second role of packages is

as a visibility modifier somewhere between private and public.

13.3.2.1 Packages and Names

A class or interface is declared to be in particular package *packageName* if the first non−blank non−comment line in the file says

```
package packageName;
```

`packageName` may be any series of Java identifiers separated by periods, such as java.awt .event and cs101.util. By convention, package names are written entirely in lower case. A file that is not declared to be in a specific package is said to be in the default package, which has no name.

Every Java class or interface actually has a long name that includes its package name before its type name. So, for example, String is actually java.lang.String, because the first line of the file String.java says

```
package java.lang;
```

and Console is cs101.util.Console, because it is declared in a file that begins

```
package cs101.util;
```

Any (visible) class or interface can always be accessed by prefacing its name by its package name, as in java.awt.Graphics or cs101.util.Console. If we declare the package `finance` as described above, the interface `finance.Instrument` would actually have a distinct name from the interface `music.Instrument`.

In some cases, you can also access the class more succinctly. If you include the statement

```
import packageName.ClassName;
```

after the (optional) package statement in a file, you may refer to `ClassName` using just that name, not the long (package−prefaced) name. So, for example, after

```
import cs101.util.Console;
```

the shorter name `Console` may be used to refer to the `cs101.util.Console` class. Similarly,

```
import packageName.*;
```

means that any class or interface name in `packageName` may be referred to using only its short name, unprefaced by `packageName`.

Note, however, that this naming role for packages is only one of convenience and does not provide any sort of actual encapsulation. The use of a shorter name does not give you access to anything additional. In particular, it does not change the visibility of anything. Anything that can be referred to using a short name after an import statement could have been referred to using the longer version of its name in the absence of an import statement.

There are three exceptions to the need to use an import statement, i.e., three cases in which the shorter name is acceptable even without an explicit import.

1. Names in the default package can always be referred to using their short names.

2. Names in the current package (i.e., the package of which the file is a part) can always be referred to using their short names.

3. Names in the special package `java.lang` can always be referred to using their short names.

You are not allowed to have an import statement that would allow conflicts. So, for example, you could not have both statements

```
import finance.*;
import music.*;
```

if both packages contain a type named `Instrument`. You could, however,

```
import finance.BankAccount;
import music.*;
```

since the first of these import statements doesn't shorten the name of the interface `finance.Instrument`. If you do import `finance.BankAccount` and `music.*`, you can still refer to the thing returned by `BankAccount`'s `withdraw` method as a `finance.Instrument`.

Package Naming Summary

A class or interface with name `TypeName` that is declared in package `packageName` may always be accessed using the name `packageName.TypeName`, provided that it is visible. (See the visibility summary sidebar.)

The class or interface may be also accessed by its abbreviated name, `TypeName`, without the package name, if one of the following holds:

- The class or interface is declared in the default (unnamed) package.
- The class or interface is declared in the current package, i.e., `packageName` is also the package where the accessing code appears.
- The class or interface is declared in the special package `java.lang`, i.e., `packageName` is `java.lang`.
- The file containing the accessing code also contains one of the following import statements:
  - `import packageName.TypeName;`
  - `import packageName.*;`

13.3.2.2 Packages and Visibility

The second use of packages is for visibility and protection. This use does accomplish a certain kind of encapsulation. We have already seen `private` and `public`, visibility modifiers that prevent the marked member from being seen or used or make it accessible everywhere. These two modifiers are absolute. Packages allow intermediate levels of visibility.

Between `private` and `public` are *two* other visibility levels. One uses the keyword `package`. The other is the level of visibility that happens if you do not specify any of the other visibility levels. This is sometimes called “package” visibility, although it differs from friendly visibility in other languages and, additionally, there is no corresponding keyword for it.

A member marked `protected` visible may be used by any class in the same package. In addition, it may be referenced by any subclass. It is illegal — and causes a compiler error — if something outside the package, not a subclass, tries to reference a member marked `protected` visible.

A member, class, or interface not marked with a visibility modifier is visible only within the package. It may not be accessed even by code within subclasses of the defining class or interface, unless they are within the package.

This means that classes and interfaces may be declared without the modifier `public`, in which case they can only be used as types within the package. Members may be declared without a modifier, in which case they can be used only within the package, or they may be declared `protected`, in which case they can be used only within the package or within a subclass. A non−public class or interface need not be declared in its own separate Java file.

Note, however that although a subclass may increase the visibility of a member, it may not further restrict visibility. So a subclass overriding a `protected` method may declare that method `public`, but not unmodified (package) or `private`.

There is no hierarchy in package names. This means that the package java.awt.event is completely unrelated to the package java.awt; their names just look similar.

Visibility Summary

A member, class, or interface marked `public` may be accessed anywhere.

A member marked `protected` may be accessed anywhere within the containing package or anywhere within a subclass (or implementing class).

A member, class, or interface not marked has “package” visibility and may be accessed anywhere and only within the containing package.

A member marked `private` may only be accessed within the containing class or interface.

We can use this approach to encapsulate certain aspects of our BankAccount example without making all of the relevant members private. After all, we want to protect these members from misuse by things outside the financial system (and therefore presumably outside the package `finance`), not from legitimate use by other things within the banking system.

So we might declare: public class BankAccount { ... }

and public interface Instrument { public abstract int getAmount(); public abstract void nullify(); }

but class Cash implements Instrument { private int amount; private boolean valid; protected Cash(int amount) { this.amount = amount; this.valid = true; } public int getAmount() { return this.amount; } protected void nullify() { this.valid = false; } }

This absence of the keyword `public` on the class definition means that the class `Cash` is accessible only to things inside the `finance` package. The `Cash` constructor is declared `protected`, so `Cash` may be created only from within this package. But the two methods that `Cash` implements for its interface, `Instrument`, must be `public` because you cannot reduce the visibility level declared for a method and the interface's methods are declared public.

[Footnote: The methods of a public interface must be public, but an interface not declared public may have methods without a visibility modifier.]

Unfortunately, the guarantees of packaging are not absolute. There is nothing to prevent someone else from defining a class to reside in an arbitrary package. For example, I could declare a class `Thief` in package `financial`, allowing `Thief` instances full access to the `Cash` constructor.

13.3.3 Inheritance

Inheritance can be used as a way of hiding behavior. Specifically, you can create hidden behavior by extending a class and implementing the additional behavior in the subclass. Conversely, labeling an object with a name of a superclass type has the property that it makes certain members of that object invisible.

You cannot invoke a subclass method on an object labeled with a superclass type that does not define that method, even though the object manifestly has the method. You can take advantage of this in combination with the visibility modifiers, for example creating a package−only subclass of a public class. Outside the package, instances of this subclass will be regarded as instances of the superclass, but because the subclass type is not available (since it is not visible outside the package), its additional features cannot be used.

For example, a specialized package−internal type of `BankAccount` might allow checks to be written:

```
class CheckingAccount extends BankAccount {
    ...

    protected Instrument writeCheck(String payee,
                                    int amount, Signatory who) {
        try {
            return new Check(payee, amount, who);
        } catch (BadCheckException e) {
            return null;
        }
    }
}
```

Now, if I have a `CheckingAccount` but choose to label it with a name of type `BankAccount`, I cannot write a check from that account:

```
BankAccount rainyDayFund = new CheckingAccount(...);
rainyDayFund.withdraw(10000);
```

works fine, but not

```
rainyDayFund.writeCheck("Tiffany's", 10000, diamondJim);
```

That is, the only methods available on an expression whose type is `BankAccount` are the `BankAccount` methods. The fact that this is really a `CheckingAccount` is not relevant.

The idea of using superclass types as ways of abstracting the distinctions between a `CheckingAccount` and a `MoneyMarketFund` is an important one. Sometimes subclasses provide extra (or different versions of) functionality. These distinctions are not necessarily relevant to the user of the class, who should be able to treat all `BankAccounts` uniformly.

Note, however, that the true type of an object is evident at the time of its construction; it must be constructed using the class name in a `new` expression. Also, if the type is visible, an explicit cast expression can be used to access subclass properties.

[Footnote: For example,

```
(CheckingAccount) rainyDayFund;
```

]

Finally, recall the discussion in chapter 10 on the inappropriateness of inheritance unless you are legitimately extending behavior. Inheritance should not be used, for example, when you need to "cancel" superclass properties.

13.3.4 Clever Use of Interfaces

The discussion above of inheritance and encapsulation applies doubly for interfaces. Interfaces are a good way of achieving the subtype properties of inheritance without the requirements of strict extension. Further, an interface type cannot contain implementation, only static final fields and non−static method signatures. This means that an interface cannot divulge any properties of the implementation that might vary from one class to another or that a subclass might override. If it's in the interface, it's in every instance of every class that implements that interface.

The example in the preceding section of a `CheckingAccount` protected by subclassing are even cleaner in the case of the `Cash` and `Check` classes, which are package−local but implement the public interface `Instrument`. This means that things outside the package may hold `Cash` or `Check` objects, but will not know any more than that they hold an `Instrument`. Any methods defined by `Cash` or `Check` but not by `Instrument` are inaccessible except inside the package `finance`.

Like a superclass, the protections of an interface can be circumvented if the implementing class type is visible to the invoking code. And, as always, the true type of an object is known when you invoke its constructor.

These issues are covered further in chapters 4 and 8, on Interfaces and Designing with Objects.

13.4 Inner Classes

The final topic in this chapter is *inner classes*. Inner classes allow a variety of different kinds of encapsulation. At base, an inner class is a remarkably simple idea: An ***inner class*** is a class defined inside another. There are several varieties of inner classes, and some of their behavior may seem odd.

Because an inner class is defined inside another class, it may be protected by making it invisible from the outside, for example by making it private. This makes inner classes particularly good places to hide implementation. The actual types of private inner classes are invisible outside of their containing objects, making the inheritance and interface tricks of the previous section more powerful.

Conversely, inner classes can also be used to protect their containing objects. An inner class lives inside another object and has privileged access to the state of this “outer” object. For this reason, inner classes can be used to provide access to their containing objects without revealing these outer objects in their entirety. That is, an inner class's instance(s) can (perversely) be used to limit access to its containing class.

Beware: Although an inner class is defined inside the text of another class, there is no

particular subtype relationship established between the inner and outer classes. For example, an inner class normally does not extend its containing (outer) class.

13.4.1 Static Classes

A ***static inner class*** is declared at top level inside another class. It is also declared with the keyword `static`. Static inner classes are largely a convenience for defining multiple classes in one place. A static class declaration is a static member of the class in which it is declared, i.e., it is similar to a static field or static method declaration.

Understanding static inner classes is quite straightforward. There are only a few real differences between a static inner class and a regular class. First, the static inner class does not need to be declared in its own text file, even if it is public. In contrast, an ordinary public class must be declared in a file whose name matches the name of the class. Second, the static inner class has access to the static members of its containing class. This includes any private static methods or private static fields that the class may have.

The proper name of a static inner class is *`OuterClassName.InnerClassName`*.

Beware: This naming convention looks like package syntax (or field access syntax), but it is not.

The constructor for a static class is accessed using the class name, i.e.,

```
new OuterClassName.InnerClassName()
```

perhaps with arguments as with any constructor.

13.4.2 Member Classes

A ***member class*** is defined at top level inside another class, but without the keyword `static`. A member class declaration is a non−static member of the class in which it is declared, i.e., it is similar to a non−static field or method declaration. This means that there is exactly one inner class (type) corresponding to each instance of the outer class. If there are no instances of the outer class, there are in effect no inner class types. When an outer instance is created, a corresponding inner class (i.e., factory) is created and may be instantiated. Note that this does not necessarily make any inner class instances; it just creates the factory object. The inner class and all of its instances have privileged access to the state of the corresponding outer class instance. That is, they can access members, including private members.

An example may make this clearer. Suppose that we want to have a Check class corresponding to each CheckingAccount. The Check class that corresponds to my CheckingAccount is similar to the Check class that corresponds to your

CheckingAccount, but with a few differences. Specifically, my Check class (and any Check instances I create) should have privileged access to my CheckingAccount, while your Check class should have privileged access to *your* CheckingAccount. So, in effect, the Check class corresponding to my CheckingAccount is different from the Check class corresponding to your CheckingAccount. It differs precisely in the details of the particular CheckingAccount to which it has privileged access. Creating a third CheckingAccount — say, Bill Gates's CheckingAccount — should cause a new kind of Check, Bill Gates's Checks, to come into existence. These Checks differ from yours and mine. Note that creating Bill Gates's CheckingAccount also creates Bill Gates's Check type, but doesn't necessarily create any of Bill Gates's Check instances. Bill still has to write those...

```
class CheckingAccount extends BankAccount {
    ...

    protected class Check implements Instrument {

        private BankAccount originator = CheckingAccount.this;
        private String payee;
        private int amount;
        private boolean valid;
          ...

        protected Check(String payee, int amount, Signatory who) {

            if (! who.equals(CheckingAccount.this.owner)) {
                throw new BadCheckException(this);
            }

            this.validate(Signatory);
            this.payee = payee;
            this.amount = amount;
            this.valid = true;
        }

        Instrument cash() throws BadCheckException {

            if (! this.valid) {
                throw new BadCheckException(this);
            }

            Instrument out = this.originator.withdraw(this.amount);
            this.nullify();
            return out;
        }
    }
}
```

In this case, there is in effect one Check class for each CheckingAccount. This is precisely what you'd want: each CheckingAccount has a slightly different kind of Check, varying by who is allowed to sign it, etc.

The proper name of a member class is `instanceName.InnerClassName`, where `instanceName` is any expression referring to the containing instance. So a way to name Bill's check type is `gatesAccount.Check` (assuming `gatesAccount` is Bill's CheckingAccount), and he can write a new Check using

```
new gatesAccount.Check(worthyCharity, 1000000, billSignature)
```

Note that he can't just say `new Check(...)`, because that leaves ambiguous whether he's writing a check from his account or from mine.

There is a special syntax that may be used inside the inner class to refer to the containing (outer class) instance: `OuterClassName.this`. For example, in the Check constructor code above, a particular Check's Signatory is compared against the owner of the containing CheckingAccount by comparing it with the owner of the containing CheckingAccount instance. This ensures that I can't sign a Bill Gates Check, nor he one of mine. It is accomplished by looking at `CheckingAccount.this`'s owner field. Note the use of the `CheckingAccount.this` syntax to get at the particular CheckingAccount whose Check class is being defined.

The Check serves as a safely limited access point into the CheckingAccount. For example, each Check knows its CheckingAccount's owner. When a new Check is being created, the Check's Signatory is compared against the account owner (CheckingAccount.this.owner, a field access expression) to make sure that this person is an authorized signer. The identity of the allowable Signatory of the check is hidden, but it is fully encapsulated inside the Check itself. Anyone can get hold of the Check without being able to get hold of the Signatory (or BankAccount balance) inside.

13.4.3 Local Classes and Anonymous Classes

There are two additional kinds of inner classes, local classes and anonymous classes. They are briefly explained here but their intricacies are beyond the scope of this chapter.

A local class declaration is a statement, not a member. A local class may be defined inside any block, e.g., in a method or constructor. There is in effect exactly one local class for each execution of the block. For example, if a local class is defined at the beginning of a method body, there is one local class type corresponding to each invocation of the method, i.e., the class depends on the invocation state of the method itself.

The syntax of a local class method is much like member class declaration, but the name of a local class may only be used within its containing block. A local class's name has the same visibility rules as any local name, i.e., its scope persists from its declaration until the end of the enclosing block. You may only invoke a local class's constructor with a `new` expression within this scope. You may return these instances from the method or otherwise use these instances elsewhere, but their correct type will not be visible elsewhere. Instead, you must refer to them using a superclass or interface type.

A local class has privileged access to the state of its containing block as well as to the state of its containing object (class or instance). The local class may access the parameters of its containing method, as well as any local variables in whose scope it appears, provided that they are declared final. If a local class is defined in a nonstatic member (method or constructor), the local class's code may access its containing instance using the `OuterClassName.this` syntax. If a local class is defined in a static member (e.g., in a static method), the local class has only a containing class, not a containing instance.

An anonymous class declaration is always a part of an anonymous class instantiation expression. Anonymous classes may be defined and instantiated anywhere where an instantiation expression might occur. They have a special, very strange syntax. An anonymous class is only good for making a single instance as an anonymous class declaration cannot be separated from its instantiation. Anonymous classes are a nice match for the event handling approaches of the Event Delegation chapter.

The syntax for an anonymous class declaration−and−instantiation expression is

```
new TypeName() {memberDeclarations}
```

where `TypeName` is any visible class or interface name and *memberDeclarations* are non−static field and method declarations (but not constructors).

[Footnote: If there is necessary instance−specific initialization of an anonymous class, this may be accomplished with an instance initializer expression. Such an expression is a block that appears at top level within the class and is executed at instance construction time.]

If `TypeName` is a class, the anonymous class extends it; if `TypeName` is an interface, the anonymous class implements it. In either case, *memberDeclarations* must include any method declarations required to make an instantiable (sub−)class. The evaluation rules for this expression create a single instance of this new — and strictly nameless — class type. Like a local class, the anonymous class's code may access any final parameters or local variables within whose scope it appears, and may use `OuterClassName.this` to refer to its containing instance if its declaration/construction expression appears within a non−static member.

Inner Classes

	Type Name (i.e., how to refer to the type)
Static Inner Class	`OuterClass.InnerClass`
Member Class	`OuterInstance.InnerClass`
Local Class	`InnerClass`
Anonymous Class	None

	Type Name Accessibility (i.e., where you may refer to the type)
Static Inner Class	Like static members (public, protected, private, etc.)
Member Class	Like non−static members (public, protected, private, etc.)
Local Class	Like local variables, i.e., only within the enclosing block
Anonymous Class	Invisible

	Class is contained within:
Static Inner Class	Outer class
Member Class	Instance of the outer class
Local Class	Block
Anonymous Class	Expression

	Does the inner class have access to static members of the containing class?
Static Inner Class	Yes
Member Class	Yes
Local Class	Yes
Anonymous Class	Yes

	Does the inner class have access to the instance of the containing class (including its fields and methods)?
Static Inner Class	No
Member Class	Yes, by using `OuterClass.this`
Local Class	Yes, by using `OuterClass.this`
Anonymous Class	Yes, by using `OuterClass.this`

	Does the inner class have access to the parameters and local variables of the containing block?
Static Inner Class	No
Member Class	No
Local Class	Yes, if they are declared `final`
Anonymous Class	Yes, if they are declared `final`

	Syntax for declaring:
Static Inner Class	*`visibility`* `static class` *`ClassName`* `{` *`members`* `}`
Member Class	*`visibility`* `class` *`ClassName`* `{` *`members`* `}`
Local Class	`class` *`ClassName`* `{` *`members`* `}`
Anonymous Class	Only possible in instantiation (see below)

	Where declared?
Static Inner Class	At the top level in OuterClass
Member Class	At the top level in OuterClass
Local Class	A statement inside a block (including method or constructor)
Anonymous Class	In an anonymous class instantiation expression

	Syntax for instantiation:
Static Inner Class	`new` *`OuterClass.InnerClass`*`( ... )`
Member Class	*`new OuterInstanceExpression.InnerClass`*`( ... )`
Local Class	`new` *`InnerClass`*`( ... )`
Anonymous Class	`new` *`SuperTypeName`*`() {` *`members`* `}`

Chapter Summary

- An abstraction relies only on general properties, leaving implementation details to vary.
- Encapsulation packages up and hides those details.
- Procedural abstraction uses methods to accomplish abstraction and encapsulation.
- A method should be short, have a succinctly summarizable function, and not contain code that is redundant with other methods.
- Abstraction and encapsulation enhance the readability, comprehensibility, modifiability, and maintainability of code.
- Packages provide grouping among interrelated classes.
- The full name of a class or interface is prefaced by its package name.
    - ♦ Import statements allow you to circumvent this longer name.
    - ♦ Some other short names are automatically available, even without an import statement.
- Visibility modifiers limit access to class members, including inner classes. Together with the use of superclass or interface type names, they provide a way to limit access to an object.
- Inner classes are a mechanism for defining one class inside another.
    - ♦ This can be used to hide the inner class.
    - ♦ This can also be used to limit access to the outer class by distributing the inner class instead.

Exercises

Chapter 14

Intelligent Objects and Implicit Dispatch

Chapter Overview

- How can I exploit method “ownership” to make objects do what I want?
- How do I pass behavior around?
- How do I know which method will be invoked?

Methods belong to objects. In some cases, as when getter and setter methods allows access to an object's internal state, the reason for housing methods in objects is clear. But in many cases, it may be less obvious why a method ought to be affiliated with a particular object. In this chapter, we look at several cases in which methods are used in concert with their owning objects to accomplish tasks that might not be obvious.

Methods can be used as a way to create implicit dispatch. Many objects, belonging to many different classes, can each be given a method of the same name (and footprint). In this case, dispatching to the correct code is as simple as asking the object to perform this method for you.

Fixing the name of the method but leaving the owning object to vary allows you to do a wide range of things. You can, in effect, pass a method as an argument (by passing its containing object), return a method from a procedure (by returning its containing object), or store it in a name or other structure (by storing its containing object). You can remember who called you and arrange to call that object back; you can build complex

---

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

homogeneous structures by exploiting the fact that one object is associated with other, equally intelligent, objects that can cooperatively solve problems that none could solve individually.

Each of these mechanisms works because every method is associated with an object. If the method name is fixed at the time that the program is written, its target object can be allowed to vary, allowing a runtime decision as to which piece of code — which instructions, which method body — should actually be executed.

Objectives of this Chapter

1. To understand Java's method dispatch mechanism.

2. To be able to use the same−named method in different classes of objects to create an implicit dispatch.

3. To appreciate how Runnables can be used to encapsulate procedures.

4. To learn how to set up and use callbacks so that a method can convey information to its calling object without returning.

5. To recognize various forms of recursion and to be able to use structural recursion as a problem−solving technique.

6. To understand, recognize, learn how, increase familiarity, master details, appreciate, discover, be able to ...

14.1 Procedural Encapsulation and Object Encapsulation

In the previous chapters, we saw how a central control loop can be used as a dispatcher, invoking different methods at different times depending on circumstances such as the value of a particular piece of state. We also saw how the different responses can be packaged up inside methods and how these methods in turn can be encapsulated inside objects. In this chapter, we will take these ideas one step further and use Java's method dispatch mechanism (plus some clever design) to determine what response is appropriate under various circumstances.

Before we turn to the use of objects as a dispatch mechanism, let's briefly review some of the properties of methods and of objects.

A method is a set of instructions to be followed. The method instructions are executed when an instruction−follower evaluates a corresponding method invocation expression, i.e., a call to the method. The method instructions may require some information to be able to execute; these are the arguments to the method. The method instructions may also produce some information; this is the method's return value.

Every method belongs to a particular object; there are no methods "just floating around" in Java. Each method body is textually contained in a class definition. Regular methods belong to individual instances of the class in which they are textually contained. (Static methods belong to the class object itself.)

For example, if `yesBox` is a Checkbox and you want to find out whether `yesBox` is currently selected, you can ask `yesBox` to supply you with that information by using the method invocation expression `yesBox.isSelected()`. There's no way to just ask `isSelected()`, though: you have to know whose `isSelected()` method it is.

Methods encapsulate behavior, but they do not by themselves encapsulate state. This is the role of objects. An object typically contains both methods — sets of instructions — and persistent information. For example, the Checkbox named by `yesBox` has a method called `isSelected()`, which provides instructions for how to determine whether `yesBox` is currently checked. When the expression `yesBox.isSelected()` is evaluated, those instructions are executed and the desired information is produced. But when the method is not being invoked, the method itself doesn't have any information or action. In contrast, even in the absence of any method invocation, the Checkbox `yesBox` contains state indicating whether it is currently selected, perhaps in the form of a private boolean field.

Objects, then, package both behavior (in the form of methods that can be invoked) and persistent state that provides a background context for that behavior. (Presumably `yesBox.isSelected()` behaves differently depending on whether the hypothetical private boolean field is true.) An object exists even when none of its methods is being invoked, and its fields persist between method invocations. An object is thus a powerful mechanism for modeling parts of the world. By making that state internal to the object, hiding it from external access, and providing a set of methods that give selective access to that state, objects can be used to encapsulate the coherent behavioral aspects of real−world things. The method `isSelected()` by itself would have little meaning. The object `yesBox` provides a context for the `isSelected()` method, so that it legitimately models coherent persistent behavior.

The name of a method to be invoked must be chosen at the time that you are writing your program. In contrast, the particular object whose method will be invoked need not be known until the time that you actually run the program. For example, when the expression `yesBox.isSelected()` is written, the method name — `isSelected` — and even its footprint — no arguments — is already known. No other method can be invoked with this expression. But at the time that the expression is written, it may not be

possible to tell to which object the name `yesBox` will refer. It may, in fact, not even be possible to tell the exact type of the object to which `yesBox` refers, although we know that it will be some type of Checkbox. (It could be any subtype of Checkbox.)

In the remainder of this chapter, we will see how fixing the method name and allowing its target object to vary gives the programmer a great deal of additional power. In the first — central — example of this technique, we will shift specialized behavior from their previous location in the handler methods within a single object to a new role within separate objects, objects encapsulating both those handler methods and associated state. We will see how this migration of behavior from procedural encapsulation to object encapsulation provides a different model for dispatch, and how it can be used to make object−oriented programming a remarkably powerful technique.

14.2 From Dispatch to Objects

Consider the following problem: You are writing code that will retrieve objects, one at a time, and print them out to the user. Some of these objects will be Strings. Some of the objects will be Points, items representing two−dimensional coordinates. A Point object has methods to retrieve its individuate coordinates, called getX() and getY(), each returning an int. And some of the objects will be Dimensions, items representing two−dimensional extents, with int−returning getWidth() and getHeight() methods. Your job is to write the printObject method.

14.2.1 A Straightforward Dispatch

You might implement this by using a simple dispatch mechanism. Since this dispatch is done on the basis of the object's class, you cannot use a `switch` statement. So we'll try an `if`:

```
public void printObject(Object o) {
    if (o instanceof String) {
        Console.println(o);
    } else if (o instanceof Point) {
        Point p = (Point) o;
        Console.println("Point: ("
                        + p.getX()
                        + ","
                        + p.getY()
                        + ")");
    } else if (o instanceof Dimension) {
        Dimension d = (Dimension) o;
        Console.println("Dimension: ("
                        + d.getWidth()
                        + ","
                        + d.getHeight()
                        + ")");
    }
}
```

[Footnote: This code suffers from a few problems, not the least of which is that it doesn't do anything about the possibility that *o* is none of the above. While we'd never write such code in a real application, we'll skip the else error condition clause here for pedagogic succinctness.]

14.2.2 Procedural Encapsulation

Of course, knowing what we do about procedural encapsulation, this looks like a superb opportunity to break out the concisely describable code. There are two relatively obvious routines lurking here:

```
public String pointToString(Point p) {
    return "Point: ("
           + p.getX()
           + ","
           + p.getY()
           + ")";
}
```

and

```
public String dimensionToString(Dimension d) {
    return "Dimension: ("
           + d.getWidth()
           + ","
           + d.getHeight()
           + ")";
}
```

We might also, for symmetry, add

```
public String stringToString(String s) {
    return s;
}
```

although it doesn't seem particularly well−motivated at the moment.

Given these routines, we might rewrite printObject as

```
public void printObject(Object o) {
    if (o instanceof String) {
        Console.println(this.stringToString((String) o));
    } else if (o instanceof Point) {
        Console.println(this.pointToString((Point) o));
    } else if (o instanceof Dimension) {
        Console.println(this.dimensionToString((Dimension) o));
    }
}
```

14.2.3 Variations

The new printObject still has a certain amount of redundant code. We can pushing the Console.println out of the individual ifs, but then we'll need to remember the String returned by each toString method. We could write

```
public void printObject(Object o) {
    String s = "";
    if (o instanceof String) {
        s = this.stringToString((String) o);
    } else if (o instanceof Point) {
        s = this.pointToString((Point) o);
    } else if (o instanceof Dimension) {
        s = this.dimensionToString((Dimension) o);
    }
    Console.println(s);
}
```

In yet another optimization, we could actually transfer the coercion into the individual toString methods, calling them on Objects rather than on specialized types. This makes the methods somewhat less general — what if they're called on the wrong type of objects? — but if we can be sure that they'll always be called appropriately, it cleans up our dispatch code further.

```
public String pointToString(Object o) {
    Point p = (Point) o;
    return "Point: ("
           + p.getX()
           + ","
           + p.getY()
           + ")";
}

public String dimensionToString(Object o) {
    Dimension d = (Dimension) o
    return "Dimension: ("
           + d.getWidth()
           + ","
           + d.getHeight()
           + ")";
}

public String stringToString(Object o) {
    return (String) s;
}
```

Now the dispatch routine reads

```
public void printObject(Object o) {
    String s = "";
    if (o instanceof String) {
        s = this.stringToString(o);
    } else if (o instanceof Point) {
        s = this.pointToString(o);
    } else if (o instanceof Dimension) {
        s = this.dimensionToString(o);
    }
    Console.println(s);
}
```

14.2.4 Pushing Methods Into Objects

We can take this whole approach one step further, and in doing so dramatically simplify our dispatcher code. Instead of trying to give this dispatcher object a toString method for each individual type that it might need to know about, we can put the toString methods into the individual types directly. For example, Point might have a method that says

```
public class Point {
   //...
   public String toString() {
       return "Point: ("
              + this.getX()
              + ","
              + this.getY()
              + ")";
   }
}
```

This is just the old pointToString, with a few modifications. First, note that we've eliminated the argument that pointToString needed. This is because the Point we're converting is `this`, i.e. the particular object whose toString() method is being executed. Second, we don't need a coercion. That's because if this set of instructions is being executed, it is because this (Point) object's toString() method has been called, i.e., we must be dealing with a Point. You simply can't call Point's toString() method on a Dimension (or a String).

A similar modification gives us Dimension's toString() method:

```
public class Dimension {
    //...

    public String toString() {
        return "Dimension: ("
               + this.getWidth()
               + ","
               + this.getHeight()
               + ")";
    }
}
```

And finally String's toString method is quite simple:

```
public class String {
    //...

    public String toString() {
        return this;
    }
}
```

Now, if `origin` names the Point with coordinates (0,0) and `square` names the Dimension with height 25 and width 25, `origin.toString()` returns the String "Point: (0,0)", while `extentless.toString()` returns the String "Dimension: (25,25)". Each object knows how to turn itself into a String using the toString() method provided by its class.

In point of fact, the Java class java.lang.Object has a toString() method, and so any Java object necessarily has a toString() method. In many cases, the toString() method is inherited from Object and so prints a rather ugly representation of the object. You may wish to override the toString() method of any class you expect to be printing out a lot. For example, there is a real class called java.awt.Point, but its toString() method isn't quite as succinct as the one we've given here.

14.2.5 What Happens to the Central Loop?

We have seen that writing the methods inside their respective classes makes them considerably more succinct. After all, the toString() method of Point just has to give instructions for how to print `this`, i.e., the particular Point whose toString() method is being invoked. At the time that the method is invoked, all of the relevant information is present in the ***target*** — the object whose method is invoked, i.e., `this`. But we haven't come to the best part yet.

Suppose that our types each implement their own toString() method. What, then, does the dispatcher look like?

The new dispatch code is

```
public void printObject(Object o) {
    Console.println(o.toString());
}
```

Where did the conditional go? The answer is that it is hidden inside Java's method dispatch mechanism. Java decides which toString() method to invoke by looking at the target's type.

Whenever an instruction−follower evaluates a method invocation expression, Java does a quick calculation to determine what kind of object the target — the method's owner object — is. Depending on the class of that object, Java looks up the appropriate method to invoke. (The argument types also play a role in selecting the method invoked, specifically by selecting a method whose footprint is appropriate.) This dispatch based upon the type of the target object is a simple form of ***polymorphism***. In general, polymorphism means doing different things with different types of objects.

If we move the dispatchee methods out to their respective classes, we give each kind of object its own type−specific way to respond to the request. Here, a particular — known, fixed — method name and footprint is polymorphic with respect to the target object to which it belongs. (Instances of many classes support the same method footprint. Each class provides a different implementation.) By allowing the target object to vary, we cause the same expression to invoke different pieces of code.

This approach has several benefits. First, the dispatcher becomes significantly more succinct. Second, the code that actually does the work is associated with a specific type, meaning that it doesn't have to worry about verifying type or coercion. Java does both dispatch and coercion automatically. The method is necessarily invoked on a target of the appropriate type, because the target helps to determine which method is invoked. Finally, if a new object type is to be added (e.g., to the printObject method), the particular instructions for converting it to a String can be added in the definition of the object's class; printObject no longer needs to worry about which types it is suited to handle. In fact, since toString is a method defined in the class java.lang.Object, printObject can handle any kind of Object at all.

14.3 The Use of Interfaces

In the example above, we gained great power from pushing the conversion to a String into each specific object type. Of course, any object type not supplied with its own toString() method simply inherits one from its superclass. Since java.lang.Object is the

root of the class inheritance hierarchy, each class is guaranteed to have a toString() method, if only the one defined for Object. But sometimes you will want to use polymorphism to dispatch to a method that isn't defined on java.lang.Object. What do you do then?

Consider the Calculator buttons of an earlier chapter. In that example, number buttons are supposed to display themselves on the Calculator screen, while arithmetic operator buttons are supposed to perform calculations and the clear button is supposed to erase whatever happens to be displayed. The central dispatcher of that program checked which button had been pressed and called the appropriate helper method, contained within the dispatcher object.

Precisely the same sort of logic that we applied to the object printer would work here. First, we need to define a series of object types. For example, we might have a NumberButton class whose ten instances represent the number keys, from 0 to 9. We might have an OperatorButton class, one of whose instances would represent the addition function of the calculator. And we might have a ClearButton class with a single instance corresponding to the calculator's clear key.

Each of these classes might be endowed with a buttonPressed method, to be invoked by the dispatcher when the corresponding calculator button is pressed. For example, ClearButton's buttonPressed method might say resetCalculator, while a NumberButton's buttonPressed method would invoke displayDigit. Whose resetCalculator and displayDigit methods are these? They belong to the calculator. In order to do its job, the buttonPressed method will need to be given access to the CalculatorState — an object representing what's going on inside the Calculator — as an argument.

```
public class ClearButton {

    public void buttonPressed(CalculatorState calc) {
        calc.resetCalculator();
    }
}
```

When the individual clear button's buttonPressed method is invoked, it will in turn ask the calculator to reset itself.

```
public class NumberButton {

    private final int whichDigit;

    public NumberButton(int which) {
        this.whichDigit = which;
    }

    public void buttonPressed(CalculatorState calc) {
        calc.displayDigit(this.whichDigit);
    }
}
```

Note that there are ten different NumberButton instances, and each instance will need to remember which digit it represents.

[Footnote: Once assigned, this digit doesn't change; hence, the field is declared final.]

When, for example, the 0 button's buttonPressed method is invoked, it asks its calculator to display its digit, i.e., 0. The code for other button types is similar.

When we are done writing these button types, we will need to add code to the calculator dispatcher (or to some other part of the system) that creates all of the necessary instances of these classes. We might, for example, stick these instances into an array indexed by the buttonID ints described in chapter 12. This would be a field of our animate calculator object:

```
private Object[] buttonObjects = new Object[Calculator.LAST_BUTTON_ID];
```

And then, inside the constructor for that object, we need initialization code:

```
for (int buttonID = 0; buttonID < 10; buttonID = buttonID + 1) {
    this.buttonObjects[buttonID] = new NumberButton(buttonID);
}
    // and so on for operators, clear....
```

Once we have instantiated these button types, what does the dispatcher look like? Its job will simply be to invoke the appropriate button object's *buttonPressed* method.

```
public void act() {
    int buttonID = this.gui.getButton();
    this.buttonObjects[buttonID].buttonPressed(this.calcState);
}
```

There is just one problem: this code won't compile. The array buttonObjects is an array of Objects. But most Objects don't have a buttonPressed(CalculatorState) method.

Why wasn't this a problem for the toString method of the object printer? Because each Object has a toString() method, we didn't have to do anything special to make the corresponding line of code — the invocation of the object's toString() method — work. However, if we try this trick with a method that isn't possessed by every object, we will find that our code won't compile. We can resolve this by using an interface that specifies this contract.

```
public interface CalculatorButton {
    public void buttonPressed(CalculatorState calc);
}
```

This interface gives just the information we need — the presence of a buttonPressed method that requires a CalculatorState — without saying anything about how a particular CalculatorState should respond to a button's being pressed. It leaves those aspects of the method to each class that provides an implementation for CalculatorButton's buttonPressed method.

We will also need to go back and add this interface to each of the individual calculator button classes. For example:

```
public class ClearButton  implements CalculatorButton  {

    public void buttonPressed(CalculatorState calc) {
        calc.resetCalculator();
    }
}
```

Now, we can rewrite our declaration of the buttonObjects array.

```
private CalculatorButton[] buttonObjects
         = new CalculatorButton[Calculator.LAST_BUTTON_ID];
```

Finally, our code will compile!

The calculator button is a more general example than the object printer, but both illustrate the same set of ideas. By pushing methods out of the central dispatcher object and into the classes representing distinct types of objects, we can package up the methods with the information that they need to do their jobs. We can also largely eliminate the explicit dispatcher of the chapter 12, using Java's method dispatch mechanism in its place. This

approach is very much in keeping with the philosophy of object−oriented design: keep behavior together with state encapsulated in objects.

14.4 Runnables as First Class Procedures

We have actually seen a special case of this kind of target−polymorphism−as−dispatch in our use of Animates as the instructions for AnimatorThreads. In that case, an AnimatorThread does very different things depending on the class of the particular object whose act() method it executes. In other words, AnimatorThread uses its constructor argument — the object whose act() method it is supposed to execute — to determine what it is supposed to do. The method footprint — act() — is fixed by the Animate contract. Naming this method there allows the programmer to write it explicitly into code. Remember, method names cannot be deduced and runtime, though their target objects can.

There is a similar situation in Java involving the interface Runnable (with a single method, run()) and the class Thread. A Thread is started on a particular object, and the Thread follows the instructions supplied by that object's run() method. By starting them on instances of different classes of Runnable objects, Threads can be induced to behave in very different ways. Like act(), run() exploit's Java's target−based dispatch mechanism to create different kinds of behavior.

But Runnables and run() can be used even without starting a new Thread, simply because they are fixed names for executable behavior that takes no arguments.

[Footnote: Everything said here for run() could be done with another method with a different name, but that name, too, would have to be fixed when the program is written. For no−arguments executable code, run() and Runnable make a convenient convention. If you wish to pass arguments to this procedure, you will need to define your own interface and your own method signature, as Java offers no standard conventions.]

Suppose that you want to pass a procedure around from one object to another. For example, suppose that you want to create a secret message and later, you will give that message to a decoder that will print out your secret message. One way to do this is to make the secret message a Runnable object and to use the secret message's run() method as a way for the decoder to get the message out.

```
public class SecretMessage implements Runnable {

    private String message;

    public SecretMessage(String message) {
        this.message = message;
    }

    public void run() {
        Console.println(this.message);
    }
}

public class SecretDecoder {

    public void decode(Runnable secret) {
        secret.run();
    }
}
```

Now, if we have

```
SecretMessage message = new SecretMessage("Meet me at midnight.");
```

and

```
SecretDecoder decoder = new SecretDecoder();
```

then we can try

```
decoder.decode(message);
```

which will print

```
Meet me at Midnight.
```

to the Java console. The message stays safe inside the SecretMessage as the SecretMessage is passed from method to method, stored in fields, returned from methods, and otherwise passed around the system. Because it has a run() method, that method can eventually be invoked to get the desired behavior from of the object.

In fact, by the time that this object makes it to the decoder, we might have lost track of the fact that it is a SecretMessage. Suppose that we have an object `toBeRun`, and all that we know about it is that it is a Runnable. We can still ask

```
decoder.decode(toBeRun);
```

And now we might find out, for example, that someone has replaced our message with some Fireworks:

```
public class Fireworks implements Runnable {

    private Color color;

    public Fireworks(Color color) {
        this.color = color;
    }

    public void run() {
        Console.println("Crash!  Bang!  You see "
                        + this.color.toString());
    }
}
```

Polymorphic dispatch ensures that toBeRun will print its message if it is a SecretMessage, and will explode colorfully if it is Fireworks. You do not need to know what kind of thing it is to arrange to send it to the right method; instead, Java's dispatch mechanism ensures that even when you don't know exactly what type of thing you have, the right method will be invoked.

14.5 Callbacks

A particular circumstance in which this "do the right thing" aspect of Java's method dispatch is important is called ***callbacks***. A callback is a situation in which one object has invoked a method of another, and the second object needs to get some information back to the first without returning from the method invocation. There are a few prerequisites for callbacks:

1. The invoking object must pass a reference to itself into the original invocation, or must otherwise indicate whose method is to be "called back."

2. The invoking method and the invoked method must agree upon the name of the callback method.

3. The invoked method must record the reference to the invoking object — the callback target — e.g., as a parameter to the original invocation or as a field.

4. At the appropriate occasion, the invoked method must invoke the callback method on the callback target. The fixed method name is used in this expression; the reference to the callback target is a variable.

Suppose, for example, that we have an object whose purpose is to create many separate

“web spiders”, simple programs that traverse the Internet looking for interesting information.

[Footnote: Such programs can be very useful, but you must be extremely careful in writing them. Serious disasters have been caused by web spiders that got out of control, for example creating so many spiders that the network filled up with spiders and couldn't sustain its regular traffic.]

Your original object will want to know when the spider finds interesting information. But the spider won't want to stop executing when it finds the first interesting piece of information. Instead, the spider should take the address of its sponsor with it when it goes crawling through the web, and any time it finds an interesting piece of information it should “call back” the sponsor object, giving it that information without stopping its execution.

The actual situation for a web spider is a little bit more complicated than this description because web spiders often don't run on the same computer as their sponsor and so can't make direct method calls. But we can use this idea as the framework for some code that illustrates callbacks.

```
public class SpiderStarter {

    private String interestingStuff = "";

    public void startSpider() {
        new Spider(this); // give invoked method
                          // a reference to the invoker
    }                     // i.e., the callback target

    // informationFound is the callback method.
    // It simply records the information...
    public void informationFound(String interestingItem) {
        if (this.interestingStuff == null) {
            this.interestingStuff = interestingItem;
        } else {
            this.interestingStuff = this.interestingStuff
                                  + " and also "
                                  + interestingItem;
        }
    }

    // This is a simple utility method.
    public void printInfoSoFar() {
        Console.println("I heard " + this.interestingStuff);
    }
}
```

This class provides three methods. The first starts up a Spider, telling the Spider who its sponsor is. The second provides a way for the Spider to call it back (when it finds information). The third provides a way for other objects to ask the SpiderStarter to let it know what information it has collected.

[Footnote: Strictly speaking, this code might be subject to problems if we start up more than one Spider. We really need to protect the interestingStuff using synchronization, as described in part 5 of this book. These issues don't affect the main point of this chapter, but you should be aware of them if you want to run a code example like this one.]

The definition for Spider might read

```
public class Spider extends AnimateObject {

    // where to record the callback target
    private SpiderStarter sponsor;

    public Spider(SpiderStarter who) {
        this.sponsor = who;  // record the callback target
    }

    public void act() {
        // Some code that looks for interesting stuff.
        // if you find it, call back
        this.sponsor.informationFound(interestingInfo);
    }
}
```

Now, we might say

```
SpiderStarter mamaTarantula = new SpiderStarter();
mamaTarantula.startSpider();
```

This starts a spider going. The “looking for interesting stuff” part of the Spider is missing, but we can still see how a Spider might take advantage of the callback mechanism. Since a Spider is an AnimateObject, its act() method will be executed over and over again. Each time, if it finds some interesting information, it will invoke its sponsor's informationFound method with the interesting information. But SpiderStarter's informationFound method just adds the new information to its information store and returns, so the AnimatorThread that runs the Spider AnimateObject is free to call its act() object again.

Consider trying to write Spider without the callback. SpiderStarter doesn't call a method of Spider's directly, so Spider can't return a String that way. Even if SpiderStarter did call Spider directly, mamaTarantula presumably wants the Spiders to keep going even after they find their first piece of interesting information. So it is very important that the individual Spiders have a way to get information back without stopping their own execution. This is precisely the kind of situation in which a callback is useful.

Callbacks are a very general mechanism that can be used any time one object needs to get information to its invoker without returning the information directly. They require agreement on the name of a method — perhaps specified by an interface contract — that will be used to produce the callback. Callbacks take advantage of the idea that Java's dispatch mechanism will call the appropriate piece of code. Good object encapsulation ensures that the information supplied in a callback gets to the appropriate place.

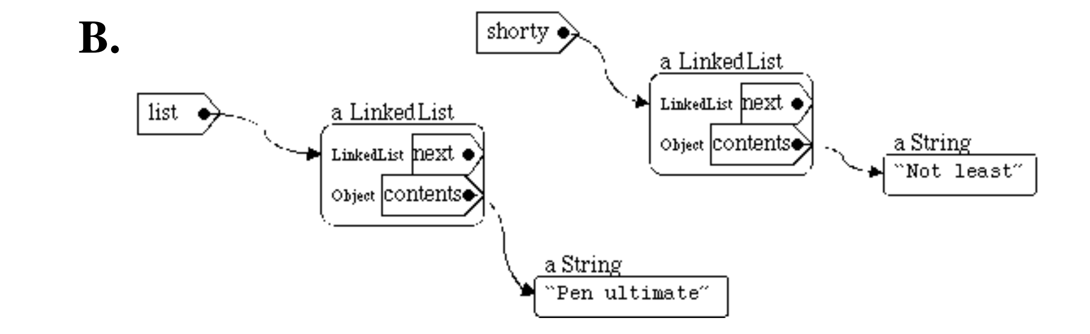
14.6 Recursion

One final example of how Java's method dispatch mechanisms work is the idea of ***recursion***. Recursion is the name for a technique in which the same named method is called over and over again, doing something slightly different each time. There are two kinds of recursion: structural recursion, which is quite common in Java and other object−oriented programming languages, and functional recursion, which is much more prevalent in functional programming languages.

14.6.1 Structural Recursion

Structural recursion is a natural extension of method dispatch to a uniform collection of objects. It is really just the idea that an object can act on its own behalf — i.e. provides methods specifying its own behavior — coupled with the idea that one object can contain — or have fields that are — other objects. For example, the calculator had (access to) many CalculatorButton objects, and it relied on them to each provide the appropriate behavior. Structural recursion is just like this, except that the object doing the relying and the component object on which it relies are instances of the same class.

A.



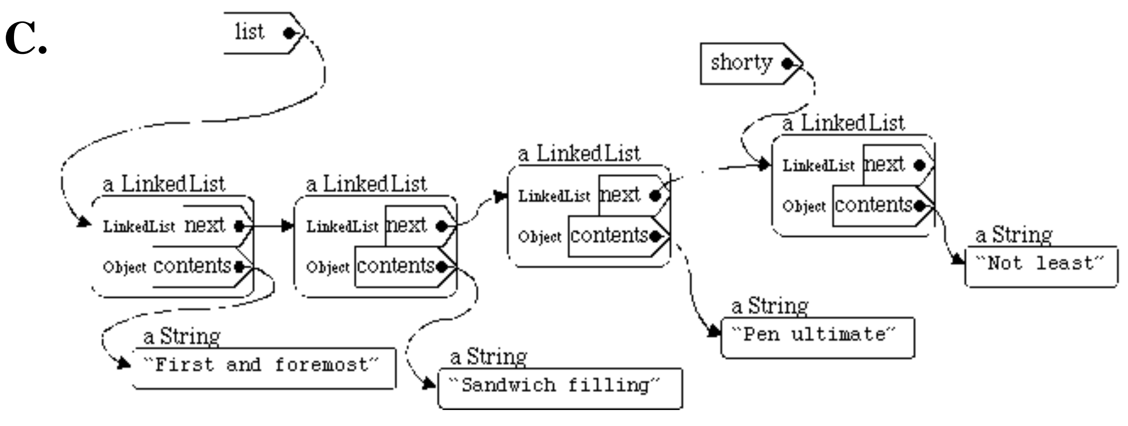


Figure 14.1. Various linked lists (following code in text).
A. After defining shorty.
B. After defining list.
C. After assigning to list.

14.6.1.1 A Recursive Class Definition

Suppose, for example, that we have a class called LinkedList:

```
public class LinkedList {

    private LinkedList  next;
    private Object      contents;

    public LinkedList(Object what, LinkedList next) {
        this.contents = what;
        this.next = next;
    }

    // maybe some methods....
}
```

To begin with, this definition is recursive. That is, the LinkedList type is defined in terms of itself. Note that this isn't at all the same thing as saying that a *particular* LinkedList is defined in terms of itself; it just means that a LinkedList consists of its contents (some arbitrary object) and its next element, which is either nothing (i.e., this is the last element) or also a LinkedList.

The idea of an object that has associates — or contains components — of the same type really isn't all that strange. For example, if we have a representation for a person, we might use the same representation for that person's parents. The same “method” for figuring out who your father is should apply equally well to figure out who *his* father is.

To create a LinkedList, you need to give it a LinkedList. To make this work, there needs to be a simple case that is not explicitly recursive. This is called a ***base case***. In the case of the LinkedList definition, the base case is `null`: `null` is a (non)value that can be associated with a name of type LinkedList that is not defined in terms of a LinkedList. A LinkedList with a `null` next field is the last element in the list.

So, for example, we can say

```
LinkedList shorty = new LinkedList("Not least", null);
```

We can also say

```
LinkedList list = new LinkedList("Pen Ultimate", shorty);
```

or even

```
list = new LinkedList("First and foremost",
                      new LinkedList("Sandwich filling", list));
```

Each of these LinkedList objects either has a next field that refers to another LinkedList object, or has a next field that is unassigned, i.e., has the value `null`.

14.6.1.2 Methods and Recursive Structure

Structural recursion is simply a way in which methods can take advantage of the recursive definition of LinkedList. It relies on the idea that each of the recursively contained objects is itself a full−fledged intelligent entity. For example, suppose that you are providing a LinkedList with a method to convert itself to a String. This method might, e.g., be suitable for printing out all of the elements contained in a LinkedList. Since one LinkedList contains another (through its next field), we can make use of the fact that that next element is also an intelligent LinkedList and will be able to convert itself to a String as well.

In writing the code to convert a particular LinkedList instance to a String, there are two possibilities.

1. Perhaps this is the last element in the list, i.e., this LinkedList object's next field is `null`. Then we can solve this problem simply: just convert the contents of this object to a String.

2. Otherwise, this is not the last element; this object contains a non−`null` next field. In this case, converting this LinkedList to a String requires converting the contents of this object, then adding a comma, then converting this object's next (LinkedList) to a String. But that next LinkedList is an intelligent object, too. We can just ask it to convert itself!

It may seem like there's a bit of sleight of hand going on here. This argument may look suspiciously like a circular definition. But it is not. Let's examine the logic here carefully.

The first of these is the simple case in which there is no further recursion. As in the definition, this is called the base case. This condition would apply if we asked the LinkedList labeled shorty to print itself — i.e., if we invoked `shorty.toString()` — which would return the String `"Not Least"`. There is only one element in this list, so printing its contents suffices.

The second case is called the ***recursive case***, the case that relies on recursion to work. It says, roughly, I know how to convert myself to a String, and my `next` knows how to convert itself to a String, so I will simply combine those two answers. Of course, the way that the `next` LinkedList element converts itself to a String relies on this same code....so here it is. Imagine this definition inside the class LinkedList, where the comment says *maybe some methods....*

```
public String toString() {
    if (this.next == null) {
        return this.contents.toString();
    } else {
        return this.contents.toString()
               + ", "
               + this.next.toString();
    }
}
```

Suppose that we invoke `list.toString()`. In this case, the object referred to by the name `list` has contents "First and foremost", so it would begin its answer with that String. But that's not enough. Because `list`'s `next` field isn't `null`, it also needs to do something about that `next` field. It can't complete its answer until it knows how to print the LinkedList that is its `next` field. Luckily, `list.next` is also a LinkedList, so it knows how to convert itself to a String. So after "First and foremost", `list` adds in a comma. Then `list` invokes its `next` field's `toString()` method to find out how to end its String.

When `list.next`'s `toString()` method is invoked, it checks to see whether *its* `next` field is `null`. Since it isn't, it can't use the base case. So it first converts its own contents into a String — "Sandwich filling" — and then adds a comma, and then asks *its* `next` field to convert itself to a String.

Once again, the LinkedList has a non-`null` `next` field, so once again the recursive case is invoked, creating "Pen Ultimate" + ", " plus the value of its `next` field's toString() method.

The `next` field of this LinkedList is the same object referred to by the name shorty. We've already seen how shorty converts itself to a String using the base case — returning "Not least" — so now we can finish off "Pen Ultimate" + ", " + "Not least". This is returned to `list.next`, completing "Sandwich filling, Pen Ultimate, Not least". Finally, this String is returned to the LinkedList labeled `list`, and that LinkedList can return its value as a String: "First and foremost, Sandwich filling, Pen Ultimate, Not least".

14.6.1.3 The Power of Recursive Structure

The power of recursion here comes from the fact that each of the individual LinkedList elements knows how to combine its `next` field's toString() with its own contents. "If only my `next` field could supply its toString()," the LinkedList seems to say, "I could produce my answer." But of course the answer for the `next` field can be constructed out of *its* contents and *its* `next` field, and so on, until we come to the base case: a LinkedList in which the `next` field is `null`, so there's no need to get its toString().

Important: It is crucial that the recursive case invoke the same−named method on a *simpler*object. That is, each recursive step must get a little bit closer to the base case. Imagine instead a situation in which you were printing a circular LinkedList. In this case, there would always be a `next` LinkedList to print, and the process would never end.

[Footnote: Actually, to prevent just such situations, the computer may have the ability to detect this circumstance — an infinite loop — and to object to it by raising an exception.]

A similar kind of structural recursion could be used to find out whether a particular object is contained in a LinkedList. In this case, there are actually two base cases.

1. If `this.contents` is the desired object, then the LinkedList contains that object, i.e., return true.

2. If `this.contents` is not the desired object, but `this.next` is `null`, then this LinkedList doesn't contain the desired object, i.e., return false.

3. Otherwise, since `this.contents` is not the desired object, this LinkedList contains the desired object exactly when the desired object is contained by the LinkedList `this.next`.

There's a fairly straightforward translation of this into Java code:

```
public boolean contains(Object what) {
    if (this.contents == what) {
        return true;
    } else if (this.next == null) {
        return false;
    } else {
        return this.next.contains(what);
    }
}
```

[Footnote: Actually, Java's && and || operators are guaranteed to evaluate their operands from left to write, proceeding only until the value of the expression is known. In the case of &&, as soon as one operand is false, no further operands need be evaluated. In the case of ||, evaluation stops as soon as an operand is true. This means that we could rewrite contains as:

```
public boolean contains(Object what) {
    return ((this.contents == what)
             || ((this.next != null)
                  && this.next.contains(what));
}
```

]

Structural recursion is an extension of “the object can handle it” to the case in which the method invocation expression is contained within the same method that it invokes. Because the target of the invoked method is a “simpler” object — one that is somehow closer to the base case — this approach ultimately produces a satisfactory answer.

14.6.2 Functional Recursion

Functional recursion is a further extension of the idea of recursion. In this case, there is no structure whose inherently recursive nature is exploited by the recursion. Instead, the necessary subsequent simplifications — steps to get closer to the base case — happen in one of the method's arguments.

For example, many kinds of numerical calculations can be performed using purely functional recursion. In this case, it is common to define one or more base cases — e.g., how the function should behave on a simple number such as 1 — and then to recursively build a solution for one number out of the solution for a smaller number. Factorial is one such function:

1. The factorial of 1 is 1.
2. The factorial of an arbitrary number, n, is n times the factorial of n−1.

The first of these is the base case. It simply produces an answer, with no recursion necessary. The second of these is the recursive case. It wishfully assumes that you know how to calculate the factorial of n−1, then uses that to construct the factorial of n. By “peeling off” one number at a time, it is possible to calculate the factorial of any number. This is really just like structural recursion, but there's no change of the method's target here.

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * this.factorial(n - 1);
    }
}
```

Factorial of 5 is 5*factorial of 4, which is 4*factorial of 3, and so on until factorial of 1, which is 1. So factorial of 2 is 2*1, and factorial of 3 is 3*(2*1), of 4 is 4*(3*2*1), and of 5 is 5*4*3*2*1. This is just like LinkedList's toString() method, except that the accumulation isn't coming from changing the target of the method invocation.

Chapter Summary

- Objects encapsulate information necessary to make methods effective.
- When multiple classes have methods with the same name, Java chooses the method that matches the target's (most specific) type.
- Dispatch can be replaced by empowering objects directly. Depending on the type of the target object, the same textual method invocation will actually call different code. This is called method polymorphism.
- A common superclass or interface, providing the method signature for the polymorphic method, is required for this kind of implicit dispatch.
- Method dispatch based on the target object can be used for other purposes as well:
    - Behavior can be passed to methods, returned from methods, and stored in objects by making it the run method of a Runnable object.
    - An executing method can give information to the object that called it, without returning, by using an explicitly agreed upon callback method.
- Recursion is a situation in which one method name is invoked repeatedly.
    - In structural recursion, the target of the method varies.
    - In functional recursion, at least one of the method's arguments varies.
    - In all recursions, there must be a base case that does not involve recursion.
    - In the recursive case, the recursive call must be to a method/target/argument that is somehow closer to the base case.

Exercises

1. Write toString() methods for an Address object and for a Date object. How would printObject have to change if it might be asked to print an Address or a Date as well as a String, Point, or Dimension?

2. Write clone() methods for Point and Dimension. (A clone() method should create a new copy of its target object.) Write a dispatcher called cloneObject(Object o).

3. Write an animate AlarmedTimer class that counts by itself, as the Timer class of chapter 9 does. In addition, it should have a setAlarm(int interval, Alarmable who) method. When this method is invoked, the AlarmedTimer should callback the Alarmable's alarmReached() method every int ticks. Here is Alarmable:

```
public interface Alarmable {
    public void alarmReached();
}
```

4. Using the LinkedList code above, add a method that returns the Object that is the `contents` of the last element in a LinkedList. For example, `list.getLast()` would return “Not least”, as would `shorty.getLast()`.

5. Define a recursive structure for a family tree. Each person in the tree should have a father and a mother, which should be either another person or — e.g., if the information were not available — `null`. Give this a method that prints all ancestors of a given individual.

   Bonus: Give this structure the ability to print only all female ancestors (using Console.println).

   Extra Bonus: Would your female−ancestor−printer print my father's mother?

Chapter 15

Event−Driven Programming

Chapter Overview

- How do we design an entity to emphasize its responses to various events?

In previous chapters, we have seen how an animate object can use its explicit control loop as a dispatcher, calling appropriate methods depending on what input it receives. In this chapter, we discuss a style of programming that shifts the emphasis from the dispatcher to the various handler methods called by that control loop. Entities designed in this way highlight their responses to a variety of situations, now called events. An implicit — behind−the−scenes — control loop dispatches to these event handler methods.

This event−driven style of programming is very commonly used in graphical user interfaces (GUIs). In Java, AWT's paint methods are an example of this kind of event−driven programming. This chapter closes with an exploration of a portion of the `java.awt` package, including `java.awt.Component` and its subclasses, to illustrate the structure of programs written in an event−driven style.

Objectives of this Chapter

1. To recognize event−driven control.
2. To understand that event handlers describe responses to events, not their causes.
3. To be able to write event handlers for simple event−driven systems.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

15.1 Control Loops and Handler Methods

In chapter 11, we looked at mechanisms for explicit dispatch. In that chapter, the job of the central control loop was to decide what needs to be done and then to call a helper procedure to do it. In this way, a single control loop can handle a variety of different inputs or circumstances. We saw, for example, how a calculator might respond differently to a digit, an operation, or another button such as =. The calculator's central control loop acts as a manager, routing work to the appropriate procedures. The actual work is accomplished by these helpers, or handler methods.

In this chapter, we will look at the same kind of architecture from a different viewpoint. Instead of focusing on the central control loop's role as a dispatcher, we will take that function largely for granted and look instead at control from the perspective of the handler methods. In other words, we will explore how one writes handlers for special circumstances, assuming that these handler methods will be called when they are needed. By the end of this chapter, we will turn to a system in which this is true without programmer effort, i.e., in which Java takes responsibility for ensuring that the handler methods are called when they are needed.

The basic idea of ***event−driven programming*** is simply to create objects with methods that handle the appropriate events or circumstances, without explicit attention to how or when these methods will be called. These helper methods provide answers to questions of the form, “What should I do when xxx happens?” Because xxx is a “thing that happens”, or an ***event***, these methods are sometimes called ***event handlers***. As the writer of event handler methods, you expect that the event handlers will somehow (automatically) be invoked whenever the appropriate thing needs dealing with, i.e., whenever the appropriate event arises.

[Footnote: Ensuring that those event handler methods will be called is a precondition for event−driven programming, not a part of it. We will return to the question of precisely how this can be accomplished later in this chapter.]

The result of this transformation is that your code focuses on the occasions when something of interest happens — instead of the times when nothing much is going on — and on how it should respond to these circumstances. An event is, after all, simply something (significant) that happens. This style of programming is called event−driven because the methods that you write — the event handlers — are the instructions for how to respond to events. The dispatcher — whether central control loop or otherwise — is a part of the background; the event handlers drive the code.

15.1.1 Dispatch Revisited

Consider the case of an Alarm, such as might be part of an AlarmClock system. The Alarm receives two kinds of signals: SIGNAL_TIMEOUT, which indicates that it is time for the Alarm to start ringing, and SIGNAL_RESET, which indicates that it is time for

the Alarm to stop. We might implement this using two methods, *handleTimeout* and *handleReset*.

```
public class Alarm {

    Buzzer bzzz =  new Buzzer();

    public void handleTimeout() {
        this.bzzz.startRinging();
    }

    public void handleReset() {
        this.bzzz.stopRinging();
    }
}
```

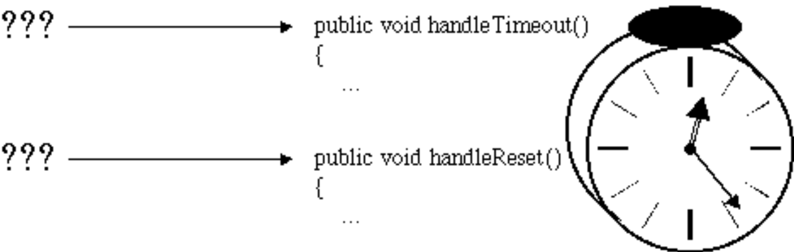


Figure 15.1. A passive Alarm object, whose methods are invoked from outside.

How do these methods get called? In a traditional control loop architecture, this might be accomplished using a dispatch loop. For example, we might make Alarm an Animate and give it its own AnimatorThread. The job of the dispatch loop would be to wait for and processes incoming (timeout and reset) signals. This AnimateAlarm's *act* method might say:

```
public class AnimateAlarm extends AnimateObject {

    Buzzer bzzz =  new Buzzer();

    public void handleTimeout() {
        this.bzzz.startRinging();
    }

    public void handleReset() {
        this.bzzz.stopRinging();
    }

    public void act() {

        int signal = getNextSignal();
        switch (signal) {
            case SIGNAL_TIMEOUT:
                this.handleTimeout();
                break;
            case SIGNAL_RESET:
                this.handleReset();
                break;
            // Maybe other signals, too...
        }
    }
}
```

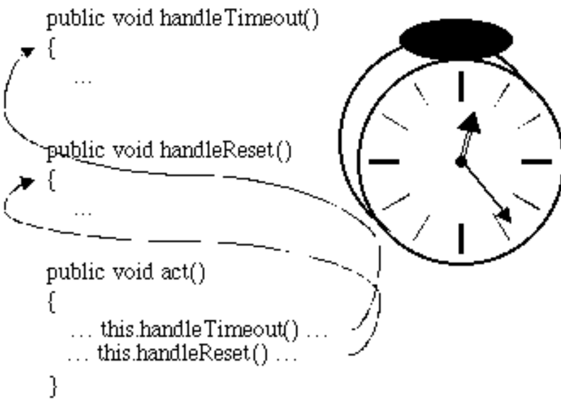


Figure 15.2. An active Alarm object, invoking its own methods.

Of course, the real work is still done by the *handleTimeout* and *handleReset* methods. The job of the dispatch loop (or other calling code) is simply to decide which helper (handler) method needs to be called. The dispatcher — this *act* method — is only there to make sure that *handleTimeout* and *handleReset* are called appropriately.

15.2 Simple Event Handling

What would happen if we shifted the focus to the helper procedures? What if we made the dispatch code invisible? Imagine writing code (such as this Alarm) in which you could be sure that the helper methods would be called automatically whenever the appropriate condition arose. In the case of the Alarm, we would not have to write the *act* method or `switch` statement above at all. We would simply equip our Alarm with the appropriate helper methods — *handleTimeout* and *handleReset* — and then make sure that the notifier mechanism knew to call these methods when the appropriate circumstances arose. This is precisely what event−driven programming does.

15.2.1 A Handler Interface

We have said that event−driven programming is a style of programming in which your code provides event handlers and some (as yet unexplained) event dispatcher invokes these event hander methods at the appropriate time. This means that the event dispatcher and the object with the event hander methods will need a way to communicate. To specify the contract between the event dispatcher and the event handler, we generally use an interface specifying the signatures of the event handler methods. This way, the event dispatcher doesn't need to know anything about the event handlers except that they exist and satisfy the appropriate contract.

In the case of the alarm, this interface might specify the two methods we've described, *handleTimeout* and *handleReset*:

```
public interface TimeoutResettable {
    public abstract void handleTimeout();
    public abstract void handleReset();
}
```

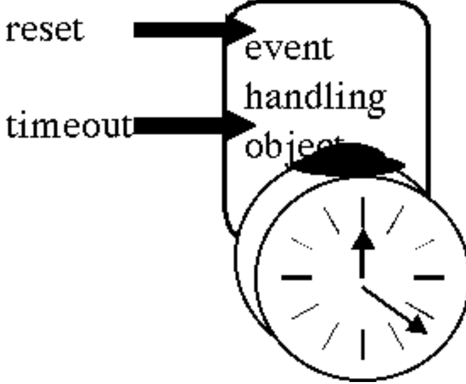


Figure 15.3. An Alarm that handles two event types.

Of course, we'll have to modify our definition of Alarm to say that it implements TimeoutResettable:

```
public class Alarm implements TimeoutResettable {

    Buzzer bzzz =  new Buzzer();

    public void handleTimeout() {
        this.bzzz.startRinging();
    }

    public void handleReset() {
        this.bzzz.stopRinging();
    }
}
```

Note that this is a modification of our original Alarm, not of the AnimateAlarm class. The TimeoutResettable Alarm need not be Animate. In fact, if it is truly event−driven, it will not be.

This TimeoutResettable Alarm definition works as long as some mechanism — which we will not worry about just yet — takes responsibility for dispatching *handleTimeout* and *handleReset* calls as appropriate. That dispatcher mechanism can rely on the fact that our Alarm is a TimeoutResettable, i.e., that it provides implementations for these methods. The dispatcher that invokes *handleTimeout* and *handleReset* need not know anything about the Alarm other than that it is a TimeoutResettable.

15.2.2 An Unrealistic Dispatcher

How might our TimeoutResettable Alarm be invoked? There are many answers, and we will see a few later. For now, though, it is worth looking at one simple answer to get the sense that this really can be done.

A simple — and not very realistic — event dispatcher might look a lot like the *act* method of AnimateAlarm. To make it more generic, we will separate that method and encapsulate it inside its own object. We will also give that object access to its event handler using the TimeoutResettable interface. Major differences between this code and AnimateAlarm are highlighted. Of course, the dispatcher doesn't have its own handler methods; its constructor requires a TimeoutResettable to provide those.

```
public class TimeoutResetDispatcher extends AnimateObject {

    private TimeoutResettable eventHandler;

    public TimeoutResetDispatcher(TimeoutResettable eventHandler) {
        this.eventHandler = eventHandler;
    }

    public void act() {

        int signal = getNextSignal();

        switch (signal) {
            case SIGNAL_TIMEOUT:
                this.eventHandler.handleTimeout();
                break;
            case SIGNAL_RESET:
                this.eventHandler.handleReset();
                break;
        }
    }
}
```

The details of this dispatcher are rather unrealistic. For one thing, it is extremely specific to the type of event, and extremely general to its event handler dispatchees. More importantly, in event−driven programming it is quite common not to actually see the dispatcher.

But dispatchers in real event−driven programs play the same role that this piece of code does in many ways. For example, the dispatcher doesn't know much about the object that will actually be handling the events, beyond the fact that it implements the specified event−handling contract. This dispatcher can invoke *handleTimeout* and *handleReset* methods for any TimeoutResettable, provided that the appropriate TimeoutResettable is provided at construction time. Different dispatchers might dispatch to different Alarms. In fact, timeout and reset are sufficiently general events that other types of objects might rely on them.

15.2.3 Sharing the Interface

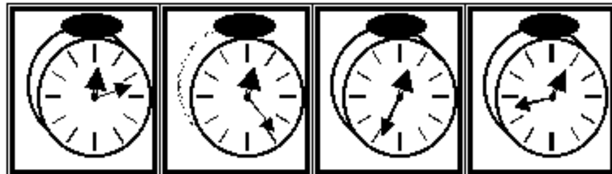


Figure 15.4. An ImageAnimation is a single component that displays a sequence of images, one at a time. For example, these frames, displayed in an ImageAnimation, would give the impression of a clock whose hands move.

Another object that might be an event−driven user of timeouts and resets — and be controlled by the TimeoutResetDispatcher — is an image animation. An image animation is a series of images, displayed one after the other, that give the impression of motion. In this case, we use the timeout event to cause the next image to be displayed, while reset restores the image sequence to the beginning. ImageAnimation simply provides implementations of these methods without worrying about how or when they will be invoked.

```
public class ImageAnimation implements TimeoutResettable {
    private Imageframes;
    private int currentFrameIndex = 0;

    // To be continued...
```

The image array `frames` will hold the sequence of images to be displayed during the animation. When the ImageAnimation is asked to paint (or display) itself, it will draw the `Image` labeled by `this.frames[this.currentFrameIndex]`. By changing `this.currentFrameIndex`, we can change what is currently displayed. When we do change `this.currentFrameIndex`, we can make that change apparent by invoking the ImageAnimation's *repaint* method, which causes the ImageAnimation to display the image associated with `this.frames[this.currentFrameIndex]`.

We omit the setup code that loads the Images into frames and handles other construction details.

The next segment of code is the timeout event handler, the helper method that is called when a timeout occurs. What should the ImageAnimation do when a timeout is received? Note that the question is not how to determine *whether* a timeout has occurred, but *what to do* when it has. This is the fundamental premise behind event−driven programming: the event handler method *will* be called when appropriate. The event handler simply provides the instructions for what to do when the event happens. When a timeout occurs, it is time to advance to the next frame of the animation:

```
public void handleTimeout() {
    if (this.currentFrameIndex < (this.frames.length - 1)) {
        this.currentFrameIndex = this.currentFrameIndex + 1;
        this.repaint();
    }
}
```

This code checks to see whether there are any frames left. If the animation is already at the end of the sequence, the execution skips the if clause and — since there is no else clause — does nothing. Otherwise — if there's a next frame — the execution increments the current frame counter, setting up the next frame to be drawn. Then, it calls *this.repaint*, the method that causes the ImageAnimation to be redrawn. Recall that the ImageAnimation paints itself using the image that is associated with **`this.frames[this.currentFrameIndex]`**.

What about a reset? What should the ImageAnimation do when it receives the signal to reset? Handling a reset event is much like handling a timeout, but even simpler. The ImageAnimation simply returns to the first image in the sequence:

```
public void handleReset() {
    this.currentFrameIndex = 0;
    this.repaint();
}
```

No matter what, we reset the current frame index to 0, then repaint the image animation with the new frame. Note also that the next timeout will cause the frame to begin advancing again.

The code to actually repaint the image, which we have not shown here, makes `this.frames[this.currentFrameIndex]` appear. As a result, *handleTimeout* works by changing the index to the next frame (until the end of the animation is reached); *handleReset* restarts the image animation by restoring the index to the beginning index of `this.frames` once more.

Both Alarm and ImageAnimation are objects written in event−driven style. That is, they implement a contract that says “If you invoke my event handler method whenever the appropriate event arises, I will take care of responding to that event.” Alternately, we think of the contract as saying “When the event in question happens, just let me know.” When building both Alarm and ImageAnimation, the question to ask is, “What should I do when the specified event happens?”

15.3 Real Event−Driven Programming

We have seen other examples of event−driven coding style. In this section, we briefly review these and recast them in light of event−driven programming's central question, “What should I do when xxx happens?” After reviewing these examples, we turn to look at the relationship of event providers to event handlers.

15.3.1 Previous Examples

In chapter 9, we saw how an Animate's *act* method is repeatedly invoked by an AnimatorThread. This *act* method is in effect an event handler. It answers the question, “What should I do when it is time for me to act?” The Animate doesn't know who is invoking its *act* method or how that invoker decided that it was time to act. It simply knows that it is, and how to respond to that knowledge, i.e., how to *act*. The *act* method may be invoked by an AnimatorThread instruction follower, executing at the same time as other parts of the system. It might equally well be invoked by a TurnTakerAnimator that controls a group of Animates and gives one Animate at a time a turn to *act*. This latter approach might make sense, for example, in a board game where each player could move only when it was that player's turn.

Similarly, we saw how a Runnable object has a *run* method that can be invoked in an event−driven style. This is commonly done when the *run* method is invoked by starting up a new Thread. In this case, the Runnable's *run* method is invoked when the Thread is *start()*ed. From the perspective of the Runnable, its *run* method is automatically invoked whenever it is time for the Runnable to “do its thing.” In a self−animating object like a Clock, *run* might be an event−handler−like method that is called by something “outside” (in this case, the Thread) when it is time for the Clock to begin execution.

The StringTransformer's *transform* methods of Interlude 1 were yet other examples of an event−driven style. These event handler methods simply answer the question, “What should I do when this StringTransformer is presented with a String to transform?” or “How do I respond to such a request?” These objects provide customized implementations for transforming strings. The decision of when to invoke these methods are outside the control of their owning objects.

In each of the cases described above, the event producer — the thing that knows that it is time for a handler method to be invoked — and the event handler — which responds to the occurrence — communicate fairly directly. For example, the TimeoutResetDispatcher polls (or explicitly asks) for signals and then directly invokes the event handler methods of its TimeoutResettable.

15.3.2 The Idea of an Event Queue

Event−driven programming by its very nature allows a more distant relationship between

event producers and event consumers. Since the producer disavows responsibility for handling the event, it doesn't need to know or care who is taking on that responsibility; it merely needs to indicate that the event has arisen. The event handler doesn't really care where the event came from ; it just need to know that it will be invoked whenever the event has happened. This dissociation between event producers and event consumers is one of the potential benefits of programming in an event driven style.

Systems that take advantage of this opportunity to separate event producers from event handlers generally contain an additional component, called the ***event queue***, that serves as an intermediary. It is important to understand how the event queue can be used and the role that it plays as an intermediary between event producers and event handlers. Unless you are building your own event−driven system from the ground up, it is not important that you be able to build it. Generally, an event queue is provided as a part of any event−based system, and the major event−based systems in Java are no exception.

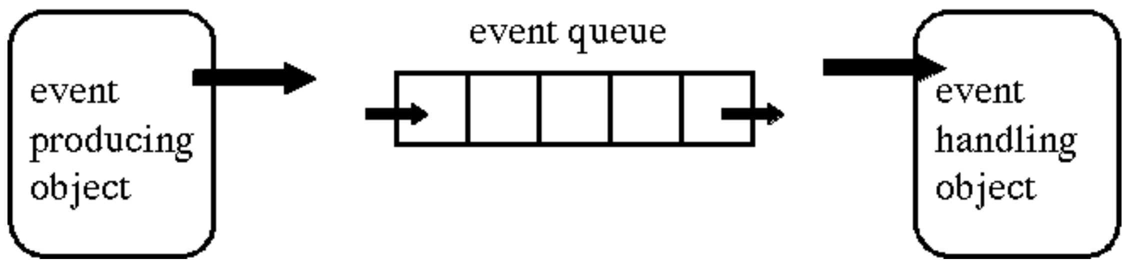


Figure 15.5. An event queue serves as an intermediary between event producers and event handlers.

The role of the event queue is to serve as a drop−off place for events that need to be handled, sort of like a To Do list. When an object produces behavior that constitutes an event, it reports that event to the event queue, which holds on to the event. The report of the event may be as simple as an indication that something happened ("Timeout!") or as complex as a complete description of the state of the world at the time that the event happened (e.g., the complete Wall Street Journal report on the stock market crash). What is important is that the event queue stores (remembers) this event report.

In addition to receiving event reports, the event queue also has an active instruction−follower that removes an event (typically the oldest one) from the queue and notifies any interested event handler methods. This is the queue−checker/dispatcher. An event queue also needs some way to figure out who to notify when an event has happened. In the cases that we explore in this chapter, there is always a single event queue per hander object, so it is always that object to which events are reported. In the next chapter, we will discuss a system that allows finer−grained control.

Consider the TimeoutResettable event handlers described above. A timer might generate the timeout events and deposit them into the queue. It would then return to its own business, keeping time and paying no more attention to the event queue. A separate instruction follower, the event dispatcher, would discover the timeout event in the queue

and invoke the *handleTimeout* method of the relevant party. The structure of this "queue cleaner" would be very similar to the TimeoutResetDispatcher we saw above.

15.3.3 Properties of Event Queues

This mechanism allows for a separation between the event producer and the event handler. The instruction−follower that puts an event into the queue — the one who generates the event — is not necessarily the instruction follower who performs the handler method (i.e., handles the event). Instead, one or more dedicated instruction followers have the task of processing events deposited into the queue, invoking the event handler method(s) as needed. Event suppliers need to know only about the event queue, not about the event handler methods.

Note that it is the event queue dispatcher's Thread (or instruction follower) that actually executes the steps of the event handler method. (Method invocation does not change which instruction follower is executing.) As a result, when you are writing event handlers, it is important that the event handler code complete and return (relatively) quickly; for example, it should not go into an infinite loop.

[Footnote: A Runnable's *run* method is an exception to this, because the Thread that executes *run* has nothing to go back to doing. When *run* completes, the execution of that Thread stops.]

If the event dispatcher invoked an event handler that did not return, the dispatcher would be unable to process other events waiting in the queue.

You will almost never have to deal with an event queue explicitly unless you write your own event−driven system from scratch. Most programmers who write event−driven programs do not ever touch the event queue that underlies their systems. Instead, like many other aspects of event−driven programming, event queueing is generally a part of the hidden behavior of a system. However, there's nothing particularly mysterious about it. An event queue's contract provides an enqueue (add to the queue) operation and a dispatcher that actually invokes the event handler methods.

[Footnote: In the next chapter, we will see that some event queues also provide an event listener registry service. This is not necessary in the event systems of this chapter, where there is a single event queue per handler object, but provides yet another layer of flexibility.]

In Java, the graphical user interface toolkit provides an event queue to handle screen events such as mouse clicking and button pressing. That event queue is fairly well hidden under the abstractions of the toolkit, so that you may not realize that it is an event queue at all. In the next chapter, we will explore that more complex system, which is used for most events in Java's windowing toolkit. That system decouples the event hander from the object to whom the event happens, allowing one object to provide the handler for

another's significant events. This is known as event delegation.

15.4 Graphical User Interfaces: An Extended Example

So far, we have left open the question of where and how events get generated. This is because in the most common kind of event system that you are likely to encounter — a windowing system for a graphical user interface — you do not deal with event generation directly. Instead, Java takes care of notifying the appropriate objects that an event of interest has occurred. When you are writing graphical user interfaces in Java, you will write event handlers without ever having to worry about when, where, and how the appropriate events are produced.

Before we can begin to talk about event handling in graphical user interfaces, we need to look briefly at what graphical user interfaces are and how they are built in Java. A graphical user interface — sometimes called a GUI, pronounced “gooey” — is a visual display containing windows, buttons, text boxes, and other “widgets.” It is common to interact with a graphical user interface using a mouse, though a keyboard is often a useful adjunct. Graphical user interfaces became the standard interface for personal computers in the 1980s, though they were invented much earlier.

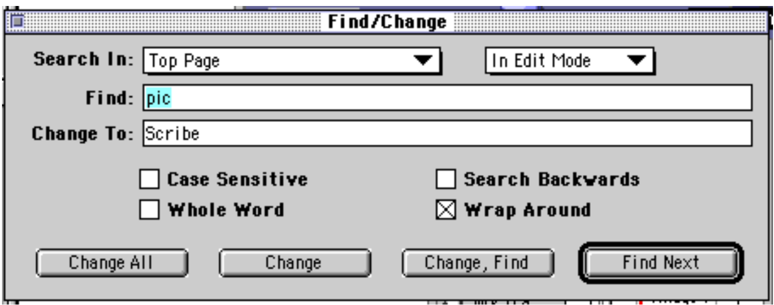


Figure 15.6. A sample graphical user interface.

[Footnote: This is a screen shot from Claris Home Page 3.0.]

15.4.1 *java.awt*

Java provides a few different ways of making graphical user interfaces. In this section, we will take a look at the package java.awt. This package contains three major kinds of classes that are useful for making GUIs. The first of these is `java.awt.Component` and its subclasses. These are things that appear on your screen, like windows and buttons. The immediately following subsection explores this component hierarchy. The second major GUI class is the class `java.awt.Graphics`, which is involved in special kinds of drawing. We will return to `java.awt.Graphics` at the end of this chapter. The final group of classes are the event classes: the `java.awt.Event` class together with

the classes in the `java.awt.Event` package. We'll come back and look at AWT Events in the event delegation chapter. Here we'll deal only with one (pseudo–)event, painting. In the remainder of this section, we are going to focus on Components and, in a bit, Graphics.

The event that we will be concerned with here is painting. That is, this is the event that occurs when a window or other user interface object becomes visible, is resized, or for other reasons needs to be redrawn. This event happens to a Component. In order to handle this event you need to know what the current state of the drawing is, including both its coordinate system and what if anything is currently visible. That information is held by a Graphics. So when the event happens, it takes a form roughly paraphrased as "paint yourself on this screen." The event handler belongs to a Component — the "self" to paint — and it takes a single argument, a Graphics — the "screen" on which to paint.

15.4.2 Components

A component is a thing that can appear on your screen, like a window or a button. The parent of all component classes is `java.awt.Component`. The Component class embodies a screen presence. You can't have a vanilla `Component`, though; you can only have an instance of one of its subclasses. (In fact, `java.awt.Component` is an abstract class. See the sidebar in Chapter 7 for further detail on abstract classes.)

Although you can't instantiate `Component` directly, `Component` has several useful subclasses. One group of these is the set of stand–alone widgets that let you interact with your screen in stereotyped ways. There are many GUI widgets built in to `java.awt`. These include `Checkbox`, `Choice`, `List`, `Button, Label`, and `Scrollbar`. In addition, there are several `Menu` variants that don't extend `Component` directly, but also provide useful widgets. Each of these widgets is pretty well able to handle its GUI behavior — showing up, disappearing, allowing selections to be made, etc. In the event delegation chapter, we will see how to use these GUI components to allow the user to communicate with your application; for example, to have something smart happen when a selection is made. (This involves customizing these widgets' event handlers.)

Another set of components are called Containers. These Components extend java.awt.Container (which itself is an abstract class extending java.awt.Component.) Containers are components that can have other components inside them. For example, a java.awt.Window (which is a kind of component) can have a java.awt.Scrollbar.

In this chapter, we will confine ourselves to one simple component behavior: painting itself. To do this, we will use a generic `Component`, called `Canvas`, that you *can* instantiate. The `java.awt.Canvas` class doesn't do anything special, but you can either use it as a generic component or `extend` it to get specialized behavior. We will make a `Canvas` that paints itself with a special picture.

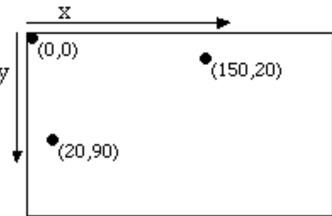


Figure 15.7. Standard screen coordinates, showing the origin, directions of increasing horizontal (x) and vertical (y) coordinates, and two other sample points.

15.4.3 Graphics

A java.awt.Graphics (sometimes called a “graphics context”) is a special kind of object that knows how to make pictures appear. A Graphics uses a coordinate system to keep track of locations within it. The origin of this coordinate system — the point (0,0) — is in the upper left−hand corner. Moving right from this point involves increasing the first (x) coordinate, so (100, 0) is 100 pixels to the right of the origin, along the top edge of the Graphics.

[Footnote: A ***pixel***, short for **pic**ture **el**ement, is the smallest visible unit on your computer's screen. A higher resolution display is one that has more pixels in the same amount of space, i.e., one with smaller pixels. Java Graphics are delineated in pixels.]

Moving down increases the second (y) coordinate, so (0,50) is 50 pixels below the top of the Graphics, along its left−hand side. (100,50) is a point that is not on either the top or left edge; it is 100 pixels to the right and 50 pixels down.

Each Graphics has methods such as *drawLine*, *fillOval*, and *setColor* that allow you to create pictures. For example, if you had a Graphics named `g`, `g.fillOval(100,100,10,10)` would make it display a 10−pixel by 10−pixel circle with its upper left−hand corner at position 100, 100. If you called `g.setColor(Color.red)` first, the circle would be red. A complete list of the methods of a java.awt.Graphics, together with a brief description of each, can be found in the java.awt.Graphics reference.

A Graphics is not the kind of object that you are likely to create or have hanging around. You will probably never run into the Graphics associated with GUI widgets or containers. However, each time that your Canvas needs to redisplay itself, it will be handed a Graphics context with which to do that redisplaying. So there will be times when your code will be given a Graphics to use.

15.4.4 The Story of *paint*

Painting (itself) is what a GUI component does when it becomes visible. For example, if a window is (partially) covering a component and then the window is moved, the

component needs to make itself look right again. Java takes care of automatically determining that this should happen and asks the component to paint itself.

Every java.awt.Component has an event−driven *paint* method. This method does not say when the component should be painted, nor why, nor on what. This method has nothing to do with determining that painting is necessary. Instead, this method is the set of instructions that describe how to paint the Component. It is the answer to the question, "What should I do when it is time to paint myself (on the provided Graphics screen)?" It is the job of whatever calls the *paint* method to determine whether and when the Component needs to be painted.

The *paint* method of a Component is passed a Graphics object. This is the Graphics which contains, among other things, the coordinate frame within which drawing on this Component should take place. It also contains a variety of utilities that will make things actually appear within the Component. Just as you don't have to determine when or whether paint should be invoked, you don't need to provide the Graphics object. Like magic, when paint is invoked, the Graphics object will be there.

Each *paint* method contains the specific instructions that that component needs to make itself appear. For example, a Button's *paint* method makes the button label appear on the button. A Window's *paint* method not only makes the Window appear, it also makes sure that the *paint* method of each of the components contained in the Window gets called as well.

When the *paint* method is invoked, it is equipped with a single argument, a Graphics. If what the Component does to display itself is, for example, to draw shapes, this Graphics (the argument passed in to the Component's *paint* method) is what actually does the drawing.

Your job, when implementing a *paint* method, is to make use of this provided Graphics (and any other information that the object may have) in order to make the correct picture appear. You supply the instructions to be executed. To paint me, make a big red dot. Or, to paint me, print my name. Or, to paint me, paint each of the Components that appear inside me.

Suppose that you want to have your Component contain a rectangle in the upper left−hand corner. A `Graphics` has a *drawRect* method which does just that. When your component's *paint* method is called, it should ask whatever `Graphics` object is supplied to it to `drawRect(int x, int y, int width, int height)`.

[Footnote: `drawRect` takes four arguments: the upper left hand coordinates and the size coordinates. All measurements are in pixels — tiny boxes that make up your screen — and the origin — the point (0,0) — is in the upper left−hand corner of the component. These are called "screen coordinates". Graphics objects have lots of other drawing methods, too. See the `java.awt.Graphics` documentation for a comprehensive

listing.]

For example, if paint were called with a Graphics named g, the instructions might read

```
g.fillRect(0,0,20,20);
```

to draw a square in the upper left−hand corner of the Component. The whole method would read

```
public void paint(Graphics g) {
    g.fillRect(0,0,20,20);
}
```

A Component's *paint* method is an event handler. This means that the Component's *paint* method is the set of instructions describing the Component's response to a request to redisplay itself. It triggers whenever Java finds that something has happened that requires the component to redisplay itself.

15.4.5 Painting on Demand

When we say that paint is an event handler method, what we mean in part is that your code doesn't call paint directly. Instead, paint is called automatically by the Java runtime system any time the Component needs to redisplay itself. This could happen, for example, if a window were covered up and then uncovered: when the uncovering event occurs, the window needs to repaint itself. Each of the components, containers, and widgets in java.awt has an event−driven *paint* method. Note, however, that there's no Paintable interface; *paint* is a method of Component and is inherited by every class that extends Component.

The *paint* method takes a `Graphics` context as an argument. You cannot, in general, supply the appropriate `Graphics` context to a `Component`; but since you don't call paint, you don't need to supply the Graphics. Instead, Java's behind the scenes bookkeeping takes care of this. (Remember, `paint(Graphics g)` is used in *event−driven* style; that is, it is called by Java, not by your program.)

Your code cannot call paint directly. It is an event handler method and it uses an event queue; only the queue manager can call paint. But sometimes you will know that it is necessary for a GUI object to repaint itself. For example, in the code above the image animation needed to repaint itself each time the `currentFrameIndex` changed. Since you can't call the component's event handler directly, each Component provides another method, called *repaint*, that you *can* call. If you call the component's *repaint* method, it will ask Java to send it a new paint event.

If you do ever need to tell the system that you want your component to be painted, you need to arrange for Java to provide the appropriate information to your class. You can do this by calling the component's *repaint* method. Unlike *paint*, which takes a `Graphics` as an argument, `repaint` takes no parameters. (This is good, because you don't generally have a `Graphics` around to give `paint`. This is another thing that Java keeps track of automatically.) You don't have to implement *repaint*; *java.awt.Component.repaint*, which you will inherit, queues up a new **`paint(Graphics g)`** request (even supplying the appropriate Graphics) behind the scenes. Remember: **You never call *paint*, and you never implement *repaint*.** To cause a painting to happen, call *repaint*; to explain how to paint your component, implement (override) the *paint(Graphics g)* method — and don't worry about the Graphics, it will be automatically supplied to you!

15.5 Events and Polymorphism

One advantage of using an event−driven style is that your code can focus on how to respond to things that happen. It does not have to spend a lot of time figuring out whether things happen or deciding what has happened and who should deal with it. (Of course, event−driven code relies on an event dispatcher, which does have to deal with these things, but often either one is available — as in the GUI case — or a fairly simple and generic one will do.)

A second advantage of the event−driven style is that, when used in concert with an event queue (as in Java's AWT), it separates the generator of the event (e.g., the window motion) from the handler of the event (the component that is uncovered). This means that these two pieces of the system can be designed independently. All they have to do is to agree on the event protocol that they will use — in this case, *repaint* and *paint(Graphics g)*. How each one fulfills its side of the contract — how the component decides to paint itself, for example — is something that the rest of the system doesn't have to worry about.

A corollary benefit, then, is that different kinds of components can handle the same event in very different ways. We saw this early in this chapter where the same pair of events — timeout and reset — were used to run both an alarm and an image animation. In these two objects, the timeout event meant very different things. The alarm handled a timeout by turning on its buzzer; the image animation switched to the next image each time a timeout occurred.

The GUI painting system that we have described uses this polymorphism to great advantage. When a component like a Canvas is asked to paint itself in a Graphics, it may draw a simple picture using the Graphics supplied. When a widget like a Button is asked to paint itself, it creates labeled region of the screen appropriate for clicking into. A Checkbox may paint itself as a square, with or without an X in it depending on whether the Checkbox is checked. A container such as a Window not only paints itself, it also asks each of the components contained inside it to paint themselves. The Window doesn't need to know anything about how these components appear; it simply asks them to paint

themselves in the way that they know best.

Chapter Summary

- By hiding the central control loop, we shift emphasis from explicit dispatch to event handler methods.

- Event driven programming separates things that happen from how they're handled.

- Each object is free to implement the same event handler in a different, customized way.

- In Java's AWT, certain GUI events are automatically dispatched by the Java runtime.

- The root of the GUI component hierarchy is `java.awt.Component`. Although `java.awt.Component` is an abstract class, it has many useful subclasses, including

    - widgets such as `Checkbox`, `Choice`, `List`, `Button, Label`, and `Scrollbar`.

    - `Containers`, `Components` that can hold other `Components`.

    - `Canvas`, a generic `Component` that you can customize.

- Every `java.awt.Component` has a *paint(Graphics g)* method that is called by Java when Java needs to make the Component (re)appear on the screen.

    - By overriding or implementing a *paint* method, you are describing how your custom component should handle requests to paint itself.

    - You don't generally call a component's *paint* method. (Among other things, you don't have a Graphics to pass it.) If you want to redraw a GUI component, you can call its *repaint* method.

Exercises

1. Define a TimeoutResettable that simply prints to the Console whenever an event happens. The message printed should differ depending on which event occurs. Implement it in a purely event driven style, i.e., assuming that something else will manage the event dispatching.

2. Describe a scenario in which an event occurs to the object in the previous exercise. Explain the sequence of action.

3. Define a class that extends `java.awt.Canvas` and has an unfilled circle with its upper left hand corner at 100, 100.

   (Bonus) What happens if you make the Canvas very small? Can you modify your class to keep the circle centered on the `Canvas`? You can use `Canvas`'s *getSize* method, which returns a `java.awt.Dimension` with directly manipulable `height` and `width` fields.

4. Define a class that extends `java.awt.Canvas` and paints itself like a black–and–white checkerboard. You may assume that the dimensions of the `Canvas` are 400x400.

   (Bonus) Make the checkerboard red and black.

5. Define a class that extends `java.awt.Canvas` and has two different painting behaviors. (For example, it could paint a black circle or a red square.) This class should also have a *changeMe* method. Each time its *changeMe* method is called, it should redisplay itself using the other behavior (e.g., it should switch between a black circle and a red square). Hook this up to a Timer (from Chapter 8).

You can test your Canvases using the cs101.awt.DefaultFrame class included in the code supplement to this book.

Chapter 16

Event Delegation and java.awt

Chapter Overview

- How do I separate an entity's core behavior (model) from its on−screen appearance (view)?
- How do intermediate (listener) objects couple together system components that don't know about one another?

In the simple event model of the previous chapter, each visible component provides an event handler method (e.g., `paint`) that is invoked every time that the appropriate event is triggered (e.g., by uncovering a window or by an explicit call to `repaint()`). The component doesn't (necessarily) have an always−active animacy (Thread); instead, it is awakened — invoked by the event dispatcher instruction follower — whenever an appropriate event occurs.

In the previous chapter, we saw how event driven programming focuses a system's design on what to do when certain events happen. The mechanism that recognizes and dispatches these events fades into the background. We saw how this approach is used to implement painting in java.awt components. In that system, each Component handles its own events. In this chapter, we will look at a more complicated two−layer model which further separates the event producer from the event consumer. This mechanism, which relies on an explicit listener registration protocol, is at the heart of the event handling system in Java's AWT versions 1.1 and later.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

The problem of GUI design is illustrative of larger design issues. The event−delegation approach described in this chapter arises from our desire to separate what happens in the GUI (such as clicking a button) from the behavior that this causes (such as playing a song). To make this work, we connect GUI objects (such as Buttons) to application objects (such as SongPlayers) indirectly, through special EventListener objects. The EventListener records the appropriate connection between GUI events and application behavior, keeping these details out of both GUI and application components. This allows significant flexibility: a single application behavior may be invoked by many different GUI events; one GUI event may give rise to many application behaviors; or the relationship between GUI events and application behavior may be remapped by a running program, for example.

This kind of indirect coupling through a Listener object is a useful technique in a wide range of applications.

Objectives of this Chapter

16.1 Model/View: Separating GUI Behavior from Application Behavior

In the previous chapter we explored event−driven programming as a way of focusing on the important things that happen in a program. An event handler is a method that responds to some important circumstance, or event. It answers the question, “What should I do when xxx happens?” It shifts the emphasis from figuring out what has happened and deciding what to do (the dispatcher) to the actual code that handles the event, whenever it may arise. Event driven programming is the idea that an object simply provides an event handler method — instructions to follow — and does not worry about how or when those instructions are executed. Somehow, an instruction−follower will invoke this method — and follow its instructions — when appropriate.

Java's AWT graphical user interface toolkit uses event−driven programming to coordinate the display of GUI objects on your computer screen. Each java.awt.Component implements its own paint(Graphics g) method, which supplies the instructions for making that Component appear in the coordinate space described by g. As in all event−driven programs, the event handler paint method does not worry about when, why, or whether it is time to paint. When the paint method is invoked, it means that the need for painting has arisen — the event has occurred — and the paint method's execution simply responds to that event.

In AWT painting, the need−to−paint event happens to a particular Component. When a need−to−paint event arises, AWT makes it clear who is responsible for handling that event: the Component that needs to be painted. But there are many other kinds of events

for which the question, “Who should handle this event?” does not have such an obvious answer. This chapter is about more general mechanisms that let programmers answer that question in a more flexible way, separating the Component to which the event happens from the object that handles the happening.

In many cases, the appearance of a GUI object and its underlying behavior may actually be implemented by two different Java Objects. For example, the GUI object that implements a set of radio buttons may be a `Panel` containing a number of ***CheckBoxes***. This is called the ***view***: what the mechanism looks like, its screen appearance. In addition, when the appropriate buttons are pressed, a song may be played. This is called the ***model***: how the mechanism behaves. The view — in this case, the `Panel` — is responsible for keeping track of the on-screen appearance of the `CheckBoxes` (with their help, of course). The `Panel` need *not* be responsible for playing the song, though. The model, which provides the song-playing behavior, may in fact be implemented by a different object. Logically, we want to separate out the GUI appearance (and GUI behavior, e.g., buttons looking pressed or not pressed) from the underlying application behavior. Java's AWT event delegation mechanism lets us do just that.

[Footnote: The event delegation mechanism described in this chapter is used in Java's AWT version 1.1 and later and also in the Java Swing toolkit. In Java's AWT version 1.0, all event handling was done using a system closer to that of chapter 15.]

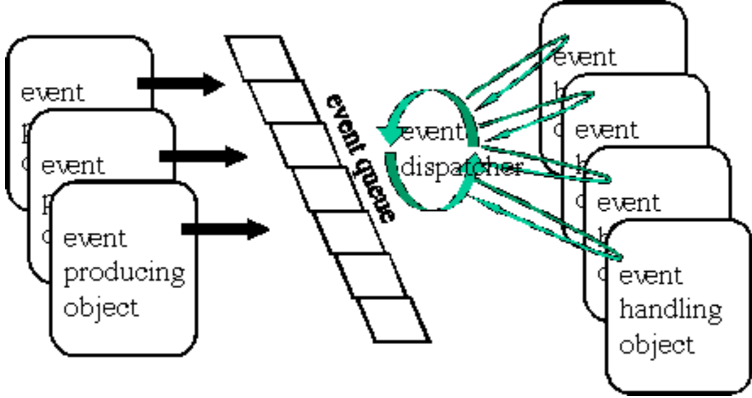


Figure 16.1. Dispatching Events: A fully general scenario.

16.1.1 The Event Queue, Revisited

In the previous chapter, we saw that we can separate the generator of an event from the actual invocation of an event handler through the use of an event queue. The event queue is a place where an event producer can “drop off” the information that an event had occurred. For example, code can call a Component's repaint() method. This adds a painting request to the event queue. Paint requests can also get added to the queue by screen events, such as a Window moving to uncover a Component or a new Window being asked to show(). Inside the queue, it doesn't matter how the event got added. A separate active event dispatcher looks at the requests in the queue and figures out which

event handlers need to be called when. The event dispatcher picks up an event (or, in the case of repaint requests, perhaps several requests) and invokes the appropriate method (e.g., paint(Graphics g)).

In the case of painting, you can imagine that there is one event queue per Component. The dispatcher doesn't need to figure out what code to call; all of the requests in that queue are for the associated Component. When a need−to−paint request arises, Java ensures that that Component's paint(Graphics) method is called. The Component doesn't have to do anything more than provide a (possibly inherited) implementation for this method.

All GUI events — not just painting — happen to particular Components. The mouse is clicked inside a particular Component. Only one Component at a time can be listening to the keyboard. (Being the Component that is listening to the keyboard is called "having the focus.") So when an event occurs, it will still get added to the queue belonging to the Component with which it is associated.

But suppose that we want to separate even event ownership from the responsibility for handling the event. Suppose, for example, that clicking a radio button (GUI Component) causes another object — a SongPlayer — to play a song. If responsibility for handling the event doesn't necessarily belong to the Component — if we are separating the Component view from a distinct Object implementing the model — the event queue's dispatcher needs to figure out who to notify that the event has occurred. We need a mechanism for associating the events that happen (and the objects to which they happen) with interested parties that are willing to handle those events. We call these interested parties ***listeners***. The system by which a separate event−handling object listens for events that occur to another (GUI Component) object is sometimes called ***event delegation***.

Java solves the "who to notify" problem by introducing the idea of listener registration. You can think of this as being something like subscribing to a newspaper clipping service or personalized online news service. When you subscribe to such a service, you give the service a list of topics that you're interested in. This is registering your interest with the event queue, or listening. The service maintains a list of subscribers along with their interests. These are the registered listeners. Each time that a new article comes in, it is added to the pile of clippings to be considered. This is putting an event into the queue. An employee of the clipping service picks up a clipping (typically the oldest one) and checks to see who might be interested. If the article matches your interests, the clipping service sends you a copy. This is dispatching to the event handler methods.

Events — such as mouse clicks or being uncovered when a Window moves — still happen to individual Components. But — for many such GUI events — each `java.awt.Component` has its own event queue that can dispatch to the appropriate registered event handlers. These event handlers need to know about and register with the Component whose events they want to listen for; they need to tell the event queue which events they are interested in handling. The Component maintains a list of listeners who

will handle its events.

Registering a listener is like leaving a (specialized) request with the clipping service: If any articles about Indonesian coffee come, please send them to Working Joe, and if any mouse motion events occur, please send them to the mouse motion listener that's waiting for them.

16.2 Reading What the User Types: An Example

Imagine that we want to have the user type her name into a GUI widget. When she does so, we will print a friendly greeting. This section walks through this example, providing a pragmatic introduction to the actual AWT mechanisms required to implement event delegation.

The code that follows assumes that it appears in a method within a class within a file that imports cs101.awt.DefaultFrame, java.awt.TextField, and java.awt.event.ActionListener, and java.awt.event.ActionEvent. In general in this chapter we will omit package names unless they are needed for clarity.

16.2.1 Setting Up a User Interaction

The first thing that we need to do is to create a place where the user can type her name. Java provides an AWT widget that is useful for just such occasions, a TextField.

```
TextField nameField = new TextField("Type your name here");
```

This line creates a TextField, a rectangular box containing text. The constructor argument is the text initially displayed in this box.

Type your name here

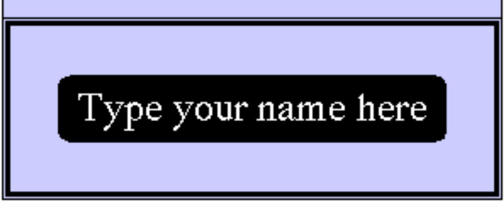
```
nameField.setEditable(true);  // Make it possible for the user
                              // to type into the TextField.
nameField.selectAll();  // Highlight the original text so that
                        // what the user types replaces it.
```

The first of these lines makes it possible for the user to type in the TextField. The second highlights all of the text in the TextField, so that what the user types will replace the text displayed there.

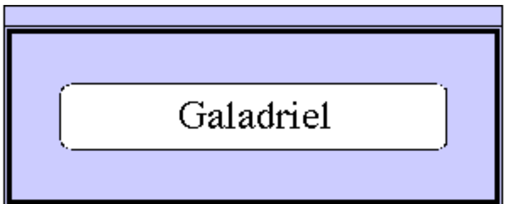
Type your name here

```
new DefaultFrame(nameField).init();  // Create a Frame
                                        around the TextField.
```

Finally, this line creates a cs101.awt.DefaultFrame, an awt Window in which a single Component can be displayed. DefaultFrame is a restricted kind of Frame, but has the advantage that it takes care of certain housekeeping details for you. DefaultFrame's init() method actually makes the window appear on the screen. See the sidebar on DefaultFrame for details.



Now suppose that the user types her name into the TextField box, replacing the highlighted text previously displayed. If the user ends her name by typing the return key, this causes an action event to be registered on the TextField. In other words, something has happened and we are ready to invoke the appropriate event handler.



Now, we are ready to print our greeting. For example, we might say

```
Console.println("Hello, " + reference_to_nameText.getText());
```

Each TextField has a getText() method that returns the String displayed in the TextField at the time of the getText() invocation. So, if we execute code along these lines, the text

```
Hello, Galadriel
```

should appear on the Java Console. There are, of course, a few issues:

1. Where does this code appear? That is, who is handling the event, and in what method?

2. How does that event handler access the TextField called `nameText` (in order to ask it to `getText()`)?

This is where Java's event delegation system comes in.

cs101.awt.DefaultFrame

A cs101.awt.DefaultFrame is a cs101 utility provided to make it easy to put up a window containing a single Component. The DefaultFrame takes care of sizing, activating the window's close box, causing the window to appear on the screen, etc.

If c is a Java component, it can be made to appear on the screen using

```
new cs101.awt.DefaultFrame(c).init();
// Create a Frame around the component.
```

The first half of this statement is an object construction expression that creates a DefaultFrame around c. The second half of the statement invokes this DefaultFrame's init() method, which is useful for its side effect: it displays Component inside the DefaultFrame, i.e., in its own window. Of course, you can use a more complex version of this code that names the new DefaultFrame, allowing you to use it elsewhere in your program, if you wish:

```
cs101.awt.DefaultFrame frame = new cs101.awt.DefaultFrame(c);
frame.init();
// Create a Frame around the component.
```

The class cs101.awt.DefaultFrame extends java.awt.Frame, documented in the AWT Quick Reference appendix to this book. For the complete code implementing cs101.awt.DefaultFrame — which is straightforward — see the online supplement to this book.

16.2.2 Listening for the Event

The event generated by Galadriel's return is associated with the TextField called nameField. That TextField is like a clipping service, and a new item of potential interest — the action taken by Galadriel — has just arrived. Now, Java needs to determine who is interested in nameField's action events.

Who might be interested? There is a special interface, called ActionListener, that describes the contract to be implemented by any object interested in handling action events. Here is the definition of the `ActionListener` interface:

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent ae);
}
```

The actionPerformed method is an event handler, so its implementation will answer the question, “What should I do when an action is performed?” In this case, the answer is to print out the text currently displayed by the TextField in which Galadriel typed her name. The object whose actionPerformed method is invoked is not responsible for deciding whether, when, or why the actionPerformed method should be called. It is only responsible for behaving appropriately when the event handler method is called.

We can build an action listener by providing a class that implements this interface. The implementation of actionPerformed in this class is an answer to the question, “What should I do when an action is performed?”

```
public class FieldHandler implements ActionListener {

    private TextField whichText;

    public FieldHandler(TextField whichTextToHandle) {
        this.whichText = whichTextToHandle;
    }

    public void actionPerformed(ActionEvent ae) {
        Console.println("Hello, " + this.whichText.getText());
    }
}
```

This class actually keeps track of which TextField it wants to associate itself with. We can create a particular FieldHandler associated with nameText using the construction expression:

```
new FieldHandler(nameText)
```

Now, when this FieldHandler's actionPerformed method is invoked — when the action happens — the FieldHandler will use nameText's getText() method to print a greeting to Galadriel.

Of course, we might want to hang on to that FieldHandler once we've created it. It will come in handy in another few paragraphs.

16.2.3 Registering Listeners

So far, so good. However, we haven't specified how the `FieldHandler` gets notified about the event in the first place. Of course, part of the story is that Java's event manager identifies that a carriage return has been hit in the TextField and generates an appropriate `ActionEvent`. But this event happens to the `TextField`; how does the `FieldHandler` get hold of it?

The answer is that Java needs to be notified that the `FieldHandler` is interested in this `TextField`'s action events. To return to our earlier analogy, the FieldHandler needs to subscribe to the TextField's action event clipping service.

This is accomplished with the `TextField`'s `addActionListener` method, which takes an `ActionListener` as an argument. The `addActionListener` method tells Java that the `ActionListener` argument `addActionListener` is wants to know about any `ActionEvents` that occur to this `TextField`. For example,

```
ActionListener nameHandler = new FieldHandler(nameText);
nameText.addActionListener(nameHandler);
```

[Footnote: or simply `nameText.addActionListener(new FieldHandler(nameText));`]

registers the `actionListener` called `nameHandler` as a listener for any `ActionEvents` that occur to `nameText`.

Now, when Galadriel finishes typing, an action event will not only be generated but also forwarded to `nameHandler` to handle.

16.2.4 Recap

The code that creates this situation is distributed over the paragraphs above. Here is the entire setup code. It might, for example, appear in a main method or in the constructor of an entity that provided the name−greeting behavior described at the beginning of this section.

```
// Set up the TextField.
TextField nameField = new TextField("Type your name here");
nameField.setEditable(true);  // Allows user typing.
nameField.selectAll();        // Highlights current text.

// Now create and register the ActionListener.
ActionListener nameHandler = new FieldHandler(nameText);
nameText.addActionListener(nameHandler);

// Finally, create a Frame around the TextField.
new DefaultFrame(nameField);
```

The only additional code required is the FieldHandler definition:

```
public class FieldHandler implements ActionListener {

    private TextField whichText;

    public FieldHandler(TextField whichTextToHandle) {
        this.whichText = whichTextToHandle;
    }

    public void actionPerformed(ActionEvent ae) {
        Console.println("Hello, " + this.whichText.getText());
    }
}
```

16.3 Specialized Event Objects

In Galadriel's example, we encountered an object whose type was ActionEvent. It appears as a parameter in the actionPerformed method of ActionListener. In that example, we blithely ignored the ActionEvent — as one often does in an action Performed method — but this begs the question of what that object is and why it appears. In this section, we'll look at ActionEvent and other similar event objects, and explore cases in which these event objects have important roles to play.

In the previous chapter, we looked at an event handler method called paint. That method needed to be supplied with a fairly specific kind of object, a Graphics, before it could do anything. In contrast, other handler methods of the previous chapter — such as handeTimeout() and handleReset() — needed no arguments at all. The event handlers in this chapter do need some information, but that information is of a fairly generic (though specializable) type. The information supplied to one of these AWT event handlers is a special Java object called an AWTEvent. Such an object inherits from `java.awt.AWTEvent` (which is itself a `java.util.EventObject`). The subclasses of `java.awt.AWTEvent` live in a separate package, called

java.awt.event.

In a general GUI, what kinds of things can happen? The mouse can be moved and clicked and dragged, the keys can be pressed, windows can be closed, menu items can be selected, text can be entered, and many, many more things can happen. A listing of the major event types used in this book may be found in the AWT Quick Reference appendix in the AWT Events segment. For example, a mouse click generates a MouseEvent, while clicking in the close box of a window generates a WindowEvent and clicking a button (or typing return in a text field) causes an ActionEvent.

Some kinds of events, like ActionEvents, are notable mostly for happening. For example, when a Button is clicked, an ActionEvent is generated. If you know what Button was clicked to generate the ActionEvent, you really know everything worth knowing about the ActionEvent. (If you don't know what Button was clicked, you can find out by asking the ActionEvent; see below.) An ActionEvent is also generated when the return key is typed in a TextField (as we have seen), indicating that the text is complete. In this case, you need to know both which TextField and, perhaps, what text was typed. But once you know what TextField generated the ActionEvent, you can ask the TextField for its text. So the internal structure of an ActionEvent is not likely to be of much interest.

Different kinds of events have methods that provide access to the different kinds of information that you'd want if you were dealing with a mouse click or a window close. These event methods are summarized in the AWT Events segment of the appendix AWT Quick Reference. For example, a MouseEvent has a few methods that are especially worth noting. If the MouseEvent is labeled mickey, then

- mickey.getX() returns an int specifying the mouse's location at the time of the MouseEvent (in pixels starting at the upper left−hand corner of mickey's screen−space). .
- mickey.getY() similarly returns mickey's y coordinate.
- If you prefer to get both coordinates at once, you can retrieve a java.awt.Point object using mickey.getPoint().

Every AWTEvent also has a getSource() method. This method returns the Object to whom the event happened. For example, we could have replaced the actionPerformed method of our FieldHandler class with the definition:

```
public void actionPerformed(ActionEvent ae) {
    TextField theField = (TextField) ae.getSource();
    Console.println("Hello, " + theField.getText());
}
```

This text uses the TextField that is the source of the action event, rather than the TextField that is handed to the FieldHandler constructor, as the target of the getText() method.

[Footnote: In this case, we could simply eliminate the constructor, making the FieldHandler definition look like this:

```
public class FieldHandler implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        TextField theField = (TextField) ae.getSource();
        Console.println("Hello, " + theField.getText());
    }
}
```

]

Some AWTEvents, such as MouseEvent, are ComponentEvents. Every `ComponentEvent` also has a `getComponent()` method that returns the same thing as its `getSource()` method, but typed as a Component.

A variety of useful event types and their methods are documented in the AWT Events segment of the AWT Quick Reference appendix.

16.4 Listeners and Adapters: A Pragmatic Detail

Every `AWTEvent` type has an associated `Listener` type.

[Footnote: Except PaintEvent, which uses the mechanism described in the previous chapter rather than the listener registration system described here.]]

This means that when the AWT event occurs — the mouse is clicked or the key is pressed, etc. — there's a type of object equipped to handle that event. (Actually, `MouseEvent` is an exception, as it has two associated listener types: ***MouseListener***, which handles clicks, entry and exit, presses and releases, and ***MouseMotionListener***, which handles drags and moves. Most event types only have one Listener.)

The `ActionListener` defined above will do the trick quite nicely for our `TextField`. The `ActionListener` interface only had a single method to implement.

Other listener interfaces are more complex, though. For example, the MouseListener interface defines five methods:

```
public interface MouseListener extends EventListener {

    public void mouseClicked(MouseEvent mickey);
    public void mouseEntered(MouseEvent mickey);
    public void mouseExited(MouseEvent mickey);
    public void mousePressed(MouseEvent mickey);
    public void mouseReleased(MouseEvent mickey);
}
```

If you want to be able to respond to mouse clicks, you will need to implement a class that has an appropriate mouseClicked method. But the MouseMotionListener interface specifies a contract with five distinct methods. If clicks are the only kind of MouseEvent that you want to respond to, it would be rather annoying to have to implement each of the other four methods just to be able to write the one (`mouseClicked`) that we need. Our class definition might say:

```
public class MouseHandler implements MouseListener {

    public void mouseClicked(MouseEvent mickey) {
        // Interesting code goes here...
    }

    public void mouseEntered(MouseEvent mickey) {}
    public void mouseExited(MouseEvent mickey) {}
    public void mousePressed(MouseEvent mickey) {}
    public void mouseReleased(MouseEvent mickey) {}
}
```

Not very concise or beautiful, but necessary if we are to implement the interface directly. After all, an interface is a contract and implementing the interface means fulfilling the whole contract, not just a part of it.

To avoid this ugliness, `java.awt.event` gives us a more concise way of saying the same thing. There is a class called `MouseAdapter` that `implements MouseListener`, providing all of the (non−interesting but also non−abstract) method bodies required. We can just `extend MouseAdapter` in our class, eliminating the need to implement all of the extra (extraneous) methods:

```
public class MouseHandler extends MouseAdapter {

    public void mouseClicked(MouseEvent mickey) {
        // Overrides MouseAdapter's mouseClicked method.
        // Interesting code goes here...
    }
}
```

Much nicer!

Each of the listener interfaces that declares more than one method has a corresponding adapter class. These are listed in the AWT Listeners and Adapters segment of the AWT Quick Reference appendix.

16.5 Inner Class Niceties

Let's return to the TextField handler class from the Galadriel example, above. There are still some improvements in functionality that we can make.

We might, for example, make our own class — our own specialized `TextField` — that is born with its own `FieldHandler`:

```
public class HandledTextField extends TextField {

    public HandledTextField() {
        ActionListener nameHandler = new FieldHandler(nameText);
        nameText.addActionListener(nameHandler);
    }
}
```

Now each HandledTextField is born with its own FieldHandler. This is similar to AnimateObject's creating its own AnimatorThread, rather than expecting someone else to create the AnimatorThread on its behalf.

Using inner classes,

[Footnote: See chapter 12 for details.]

we can make this innovation do even more work for us. Inner classes are a relatively advanced feature of Java, and they add only to the aesthetics of this program, not to its functionality. They do provide a little bit more protection for code from unanticipated use, a feature that we can exploit. After all, a FieldHandler as we have defined it is not really of much general interest. We can embed the definition of that class inside the HandledTextField class definition, hiding it from the rest of the world and simultaneously

taking advantage of inner class's privileged access to their containing instance's state.

Using inner classes, we can write:

```
public class HandledTextField extends TextField {

    public HandledTextField() {
        ActionListener nameHandler = new FieldHandler();
        nameText.addActionListener(nameHandler);
    }

    private class FieldHandler implements ActionListener {

        public void actionPerformed(ActionEvent ae) {
            Console.println("Hello, "
                            + HandledTextField.this.getText());
        }
    }
}
```

Since `FieldHandler` is defined inside `HandledTextField`, it has access to its containing instance directly (through `HandledTextField.this`), and we can eliminate the constructor argument (and the constructor itself!) for `FieldHandler`. Pretty neat, huh?

Chapter Summary

- EventListeners are interfaces promising particular sets of event handler methods. There are Listeners for groups of related AWT event types, such as mouse motion events, in the package java.awt.event.

- That package also includes adapter classes to make implementing these interfaces easier.

- Listeners are connected to AWT components using a component's `add`*`EventClass`*`Listener()` (registration) method.

- `java.awt.AWTEvent` and its subclasses are data repositories that record relevant information about individual (GUI) events.

- Each event handler method takes one of these Event objects as an argument, in much the same way that `paint()` requires a `Graphics`. Like `paint()`, the event handlers of an EventListener are called by the system, not by your code.

- Inner classes provide a nice way of packaging the definitions of subsidiary classes (such as EventListeners) inside other class definitions.

Exercises

1. Define a class that implements `java.awt.`event.MouseListener and extends `the mouseClicked(MouseEvent) method` by printing the coordinates of the point on which the mouse had clicked. You may also want to make use of the class `java.awt.event.MouseAdapter`. (Bonus: also print the components of the *previous* mouse click.)

2. Now define a class that extends `java.awt.Canvas` and sends its mouse events to your `MouseListener`.

3. Define a class that implements `java.awt.event.WindowListener` and extends the `windowClosing()method` by printing "`Nah, nah, you can't kill me!`" (Alternately, you can do the potentially more useful thing and (1) call the object's `dispose()` method and (2) call `System.exit(0)`.) What class do you think would be useful when implementing `WindowListener`?

4. Define a class that extends `java.awt.Canvas` and looks like a (black and white) Japanese flag, i.e., it has a circle at `(100,100)`. Make the circle change color when the mouse is over your Canvas. (Hint: mouse enter, mouse leave.)

Part 5

Systems of Objects

Chapter 17

Models of Communities

This chapter has not yet been written.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

Chapter 18

Interfaces and Protocols: Gluing Things Together

This chapter has not yet been written.

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

xxx

Chapter 19

Client–Server Interaction Patterns

Chapter Overview

- Who is responsible for getting something from one entity to another?
- What tradeoffs are involved in this decision?

This chapter concerns the ways in which responsibility for (information) transfer can be allocated between the provider and the recipient and the implications of these design decisions. When the service provider takes responsibility for the transfer, it maintains control of its own workload but may overwhelm the recipient. When the recipient initiates the request, the dual situation is in effect. The participants in this relationship are called servers and clients, and client/server architectures are common in modern software design.

Objectives of this Chapter

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

19.1 What Is a Client−Server Interaction?

Sometimes, one (computational) entity has something that another (computational) entity needs. For example, a baker may have cookies, and you may be hungry. In this case, the entity that has the thing—the baker—is called a ***server*** and the entity that needs the thing—you—is called a ***client***. Although these terms are often used without further explanation, you can see from this description that a client and a server are defined with respect to some (computational) need, or ***service*** (like a cookie).

In the computational world, a server is often something that provides a particular service to other computers connected by a network. For example, it is common for an organization to have a lot of disks on which its members' information is stored, and to have a single machine responsible for providing access to this storage space. This machine is called a ***disk server*** or a ***file server***. Another machine in the same organization might control the public html access for that organization's world−wide web pages. That machine would be the organization's ***web server***. Yet another machine might be in charge of electronic mail for the organization: the ***mail server***.

In each of these cases, the service is described by what is provided. But it is also important to characterize how the service is provided, in what form, when, by whom, and to whom, whether it is provided once or repeatedly, whether it is provided to one client at a time or to many clients simultaneously, and who is responsible for initiating the transaction. For example, an important message may be transmitted by certified mail, or it may be communicated by announcement over a public address system. These two services may communicate the same information, but they do so in dramatically different ways.

19.2 Postal Services: An Example

In this section, we'll look more closely at a particular real−world, non−computational service: the postal system. In doing so, we will see that most of the major properties of client−server interactions are present in familiar transactions.

The main service provided by the postal system is the transmission of physical letters and packages. In this sense, it is perhaps the original mail server. Its clients are the people who send and receive mail through the system. The post office (or, more generally, the postal system) is the server. It may be obvious that the recipient of mail is a client of the postal system: A recipient gets mail delivery from the postal system. Perhaps less intuitively, the sender of the mail is also a client (of another service of) the postal system: the sender relies on the post office to provide the service of transmitting the letter. We will return to this point later.

Figure 19.1. The post office provides parcel transmission and parcel delivery services. Its clients are senders and recipients, respectively. The post office is a server providing the service of transmission of packages.

This example is actually quite rich, illustrating several points about service providers.

19.2.1 A Server Can Provide a Variety of Services

We have already seen two interrelated services that the postal system provides: mail transmission (to mail senders) and mail delivery (to mail recipients).

Actually, each of these is itself an abstraction of several different services. For example, letter transmission is a somewhat different service than parcel transmission. The post office charges different rates depending on the weight and size of the item to be delivered. Similarly, the postal system provides multiple qualities of service for similar items. A letter can be sent air mail or surface mail, overnight or second day or standard delivery. A parcel can be sent first class or book rate. Each of these services is a specialized form of mail transmission, with different costs, time behaviors, and guarantees. All can be described as the same general mail transmission service, but each has slightly different behavior.

Some service specializations don't fit neatly under the same abstraction. For example, a letter can be sent certified; a return receipt can be requested. Requesting a return receipt even changes the contract between the client — the mail sender — and the postal system so that their interaction pattern is different. In traditional mail transmission, the client gives the item to the post office and the interaction ends. (Of course, the post office is still obliged to carry out its end of the deal, delivery.) When a return receipt is requested, the transaction does not end here. Instead, delivery involves the post office's obtaining a signature from the recipient. This signature needs to be transmitted back to the sender; only then is the original transmission service complete. This amounts to the postal equivalent of a callback. (See the chapter on Intelligent Objects.)

The same server that provides these transmission and delivery services — the post office — also provides a number of other services, some of which may not even seem related. For example, the United States Postal Service sells stamps and money orders. By special

arrangement (i.e., the rental of a post office box) it will hold your mail for you. Some post offices will even provide you with a passport. This last service is one provided by the post office acting on behalf of the Passport Agency.

When we talk about a server, then, it is important to distinguish what service that server is providing. In general, it is impossible to talk about a client or server without (at least implicitly) referring to the service provided.

19.2.2 You Can Have More Than One Provider of a Given Service

Among the services provided by the post office is the selling of money orders. But if you want to transmit money through the mail, you can also do so using a check. The check is a service provided by a bank, not by the postal system. So, for sending money through the mail, you can turn to a variety of service providers. Each will have its own set of properties: cheap or expensive, secure or less so, available on demand or only during certain hours, etc.

Figure 19.2. Multiple servers may provide similar services. Both the post office and DHL provide parcel transmission services, though these services vary in cost, guarantees of timeliness, and other properties.

Even considering only the delivery service that is the postal system's “core” service, there are still alternative providers. In the United States, package and letter delivery is provided by United Parcel Service, Federal Express, DHL, and many other vendors. Each of these service providers — servers — has a different performance profile. For example, some of these parcel delivery servers are faster, provide “better” service, include a variety of guarantees, cost more or less, etc. At different times, you may wish to select a particular server because its properties best match your needs. But even when multiple servers provide the same (or similar) services, any one service instance — such as sending one particular parcel — is likely to go through only one provider of a particular service. For example, when you mail your mother's birthday present, you will pick one carrier to deliver it.

19.2.3 Services Can Be Layered

We have seen that the post office provides both transmission and delivery services. These two services together can be used as the basis for other service models.

Figure 19.3. Layered Services. The mail order company provides the shirt service to the purchaser by means of the post office delivery system. The solid arrow shows the shirt procurement service, of which the mail order company is the server and the purchaser the client. The dotted arrows show the parcel shipping and parcel delivery services, of which the post office is the server and the mail order company and purchaser are the clients, respectively. The shirt service is implemented in terms of the parcel shipping and delivery services.

Consider a mail order company, such as a clothing vendor. If I want to buy a Hawaiian shirt, I can order it from the mail order company. The company plays the role of the server and I play the role of the client in this shirt−procurement transaction.

But the clothing vendor is not in the business of shipping. How does the clothing vendor get the shirt to me? One answer is that the clothing vendor may use the post office (or some other parcel delivery service) to ship the shirt. In this case, shirt−procurement is ***layered*** on top of parcel delivery, i.e, relies on parcel delivery to accomplish the transaction.

Real services often work this way. In fact, network services — the way that one computer communicates with another — involve many layers of services. When we look more closely at network services in chapter 21, we will examine only the highest levels of these services. We will use network transmission to build still more sophisticated services — such as a web server or a chat program — in exactly the same way that the mail order company relies on the postal service to deliver its shirt.

19.2.4 Roles Are Relative to a Service

We have seen how a single server can provide many different services (as the post office does) and that a single service may be provided by many different servers (like the various parcel delivery servers). We have also seen how layering makes it possible for

one service to be built out of others. Each of these observations provides further illustration that the role “server” (or “client”) is not an absolute one, but is meaningful only relative to a particular service.

We can't, for example, properly say that the post office is a server. We have to specify what service the post office is providing to whom. Of course, we sometimes skip this information when we think that it is obvious from context. But properly, every server is the server of a particular service interaction; every client is a client with respect to a service interaction.

This is particularly important when we're talking about sophisticated interactions in which a single entity can be simultaneously a client and a server. (Not of the same service interaction, of course.) So, for example:

- The mail order company is simultaneously the server of “shirt purchasing” (I'm the client) and the client of the post office's delivery service.
- In most standard retail transactions, the retailer is simultaneously the client of the wholesaler (who sells the retailer the goods) and the server to the general public (like me).
- Many interactions are two−way, like barter. For example, one farmer may supply eggs to a second; the second may provide the first with milk. Each farmer is a service provider (server) as well as a service consumer (client).

Note that in any relationship, an entity can either be a client or a server *of a particular service instance*, but not both.

In computational systems, we typically reserve the term server for ongoing service providers, i.e., persistent entities that can be repeatedly called upon to provide services. That is, servers are full−blown computational entities, not simply program segments.

19.3 Implementing Client−Server Interactions

As we have seen, a client−server interaction is one in which the server has something at the beginning and the client has it at the end. This “thing” might be quite abstract — permission to access some data, for example, or the property of being subscribed to a mailing list — but the idea is that the client wants it and the server can provide it.

In this section we will to focus on the question of *who initiates the service* and the implications of this decision for client−server interactions. There are many different services provided by many different servers, and many different mechanisms to support these services. These include simple procedure call, the use of channels to transmit requests, even aspects of event−driven programming. The issue of who takes responsibility for service initiation exists no matter what service mechanism is used to

implement the interaction, and the tradeoffs described here apply to each of these implementations as well.

19.3.1 Client Pull

If a client needs something (or some service) from a server, perhaps the easiest way for this transfer to happen is for the client to go and get — or ***pull*** — the thing from the server. We do this all the time. For example, this is what happens when we go to the grocery store.

- The client requests the service as it is needed.
- The server handles each request as it comes in.

The following icon represents a client pull client: In this icon, information flows from right to left. The client (the circle) initiates the transfer of information, requesting it from the server and retrieving it. Here is a client pulling from a (passive) server:

Getter methods are very simple versions of client pull. In a getter method, one entity asks another for something; the method return completes the pull request. When direct method invocation is not available — as when communicating over a channel, or network — a pull request usually consists of two separate messages: one from the client, requesting the service, and one from the server, completing it.

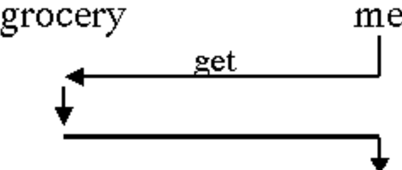


Figure 19.4. A client pull interaction: I get what I need from the grocery store. Vertical lines represent computations that happen within an entity. Horizontal lines represent communication between entities. When I ask the grocery store for something, the grocery store gives it to me.

19.3.1.1 Locating the Server

In order for a client pull interaction to work, the client must know where to find the server. This can be accomplished either by telling the client about the server when the client is created or by providing a standard place to look. For example, I might first ask the phonebook where the grocery store is, then get what I need from the grocery store. This interaction involves two separate client pulls and is depicted in the next figure. Structurally, this is the way that computers locate each other on the internet.

Figure 19.5. First I (client pull) get the grocery store's location from the phone book, then I (client pull) get the food I need from the grocery store.

19.3.1.2 Client Pull Tradeoffs

There are many advantages to client pull. The server doesn't have to do anything unless a request is pending. The client gets only what it needs, when it needs it. The client exercises control over the interaction, so the interaction (theoretically) happens when and where the client is best able to make use of the service.

On the other hand, there are disadvantages, too.

- The burden is on the client. (You have to go to the grocery store. Sometimes, you may miss out on a special because you get to the store at the wrong time.)
- The server may be deluged with requests and unable to keep up. (The grocery store may run out of something.)
- The network may be full of requests, since each client is sending these requests separately. (The check−out line may be long.)
- In general, more energy will be expended. For example, there's likely to be a lot more (network) traffic. (Each client arrives in his/her own car. Sometimes there are traffic jams in the parking lot.)
- The server's load may be patchy and unpredictable — too busy one moment, unused the next — making inefficient use of the machine and its resources. (Adequate inventory and staffing levels may be hard to identify, making the grocery store an inefficient business.)

Client pull works well when client requests are highly variable but overall not too great a load on the server. It allows each client to do its grocery shopping precisely when it needs to. When it doesn't work, though, the grocery store can be quite a mess!

19.3.2 Server Push

An alternate architecture that addresses some of these problems is for the server to take initiative. In this case, it can simply deliver — or ***push*** — the information to the client when it is ready. This is sort of like the fruit–of–the–month club. Every month, the fruit–of–the–month club delivers a box of fresh, ripe fruit to your house.

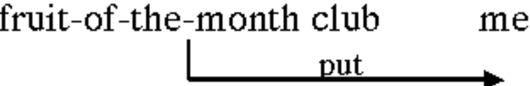


Figure 19.6. The fruit–of–the–month club delivers a box of fruit to me each month, without my having to do anything. This is server push.

In direct method invocation, setter methods are server pushes. Although these methods technically complete with a return, no value is returned; in a put, only one–way communication is necessary. In channel– or network–based interactions, a push is often implemented as a single communication.

We can represent a server push server with the icon and a server push interaction with a (passive) client with the icon . Again, information flows from left to right. In this case, however, the entity with the information initiates the service.

19.3.2.1 Registering with the Server

Before the fruit–of–the–month club can provide me with regular deliveries, however, I may need to register my interest with the club. This is often done as a one–shot communication that precedes the regular (server push) delivery. Many subscription services — like the fruit–of–the–month club, magazine subscriptions, or other periodic deliveries — require a registration before the recurrent server push can begin.

19.3.2.2 Server Push Tradeoffs

There are numerous advantages to server push approaches:

- The server gets to control who gets what when. This means that it can manage its resources and keep its load even (or at least predictable): clients with names A–G this week, H–M next; oranges this month because they're in good supply.
- If the server is supplying multiple clients with the same — or similar — information, there may be economies of scale. For example, the server may only have to package up the information once to send to multiple clients.
- The client doesn't have to do anything to make this happen. The service just shows up whenever the server thinks it appropriate.

But this model doesn't always work perfectly, either. Why not? Let's consider what can go wrong:

- Deliveries might come at a bad time. Imagine that a whole month's shipment of fruit arrives the day after you've left town for a week. By the time you get back, the fruit will have spoiled. This happens in computational systems, for example, when the server goes too fast for the client, and the client has to spend all of its time handling the server's shipments. (Sometimes, this happens even if all that the client is doing is throwing the server's information away: trash can pile up and become overwhelming. In other cases, the client *can't* throw the information away, because the next delivery depends on the previous one in some crucial way.)

- Even though the server maintains control, it can still get too busy and deliveries can get back−logged. Some clients might need more frequent attention. Such a client might not get what it needs often enough, or even in time. The fruit−of−the−month club might be reliable, but not all computational servers are.

- Sometimes, the server doesn't know that (or when) the client wants or needs something.

Both the pros and cons of this approach can be summed up by the following:

- The client has very little control over what it gets when.

- The server has lots of control, but also has to do all of the work.

One popular way of doing animation in the early days of the web involved having the web server regularly push the next image in the animation sequence. This often swamped clients — web browsers and the machines running them — making it difficult for their users to do anything at all and giving server push a(n undeservedly) bad name.

19.4 The Nature of Duals

Server push and *client pull* are opposites of a special sort. The positive aspects of one are the negative aspects of the other. In general, they are like mirror images. Pairs of opposites like this are called ***duals***, and they have some special properties. For example, you can generally take almost any statement expressed in terms of these dual operations (and their associated dual terms, such as *client* and *server*) and replace each operation with its dual without changing the truth or falsity of the statement:

- *Client pull* gives the *client* a lot of flexibility, but the *server* doesn't have much control over its workload.

Dual statement:

- *Server push* gives the *server* a lot of flexibility, but the *client* doesn't have much control over its workload.

Of course, it's not quite this simple — it's not *too* hard to find statements that you can't turn around this way — but client pull and server push *are* duals, which mean's that there's a fundamental symmetry in the ways that they work.

19.5 Pushing and Pulling Together

It is possible to build a system that uses multiple — chained — server pushes to produce its result. In this case, the client of one push becomes the server of another push: For example, a farmer may push produce to the wholesaler — taking it to market when it is ready — while the wholesaler in turn may deliver it to the retailer when it becomes available.

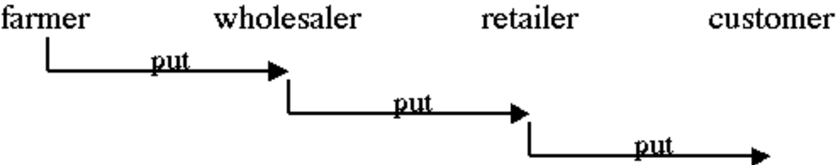


Figure 19.7. You can chain together server pushes: the farmer sells to the wholesaler, who sells to the retailer, who sells to the customer.

Similarly, a chain of client pulls — requests for services — allows one client to pull from a server that may in turn request assistance from another service: Requesting a book on interlibrary loan follows this process.

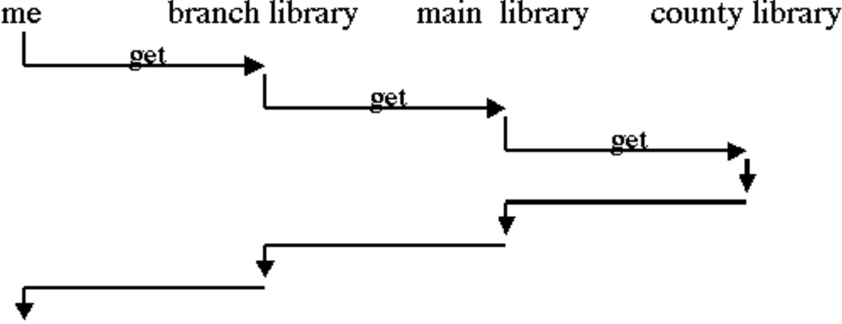


Figure 19.8. You can also chain together client pulls: I reserve a book at my branch library, which asks the main library, which sends out the request to the entire county system, which finally finds the book.

Note, however that each of these pictures involve the chaining of similar service models. It is less simple to put a client pull client together with a server push server (or a server

push client with a client pull server). (Iconically, there's no way to connect to or to .) To make these transactions possible, we need to introduce additional machinery. You cannot connect server pushes to client pulls (or client pull to server pushes — there's that dual thing again!) without putting something different in the middle.

19.5.1 Passive Repository

A passive repository is essentially just a the server side of a client pull combined with the client side of a server push. In other words, it's the passive recipient of information provided to it, and the passive provider of information when requested. It corresponds to the “drop box” where a spy might leave information for his spy master. The server — or spy — can drop off the information any time it wants. The client — or spy master — can come by and pick up the information whenever it is convenient. Iconically, this is just a . It has the important property that it can be used to connect a server push ( ) with a client pull ( ), making a functioning system:

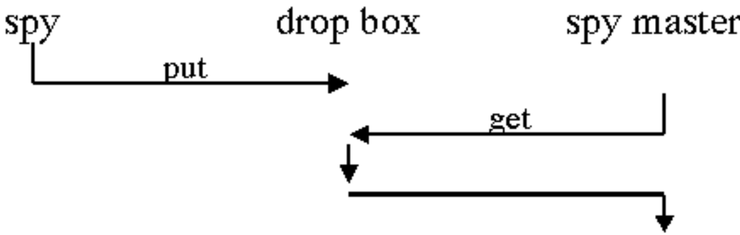


Figure 19.9. A passive repository, like a spy's drop box, accepts both server pushes and client pulls.

What happens if the server pushes a new value before the client pulls the previous one? One possibility is that the passive repository actually contains a queue, i.e., keeps track of each of the items it's been given and provides them to the client pull client, either one at a time or all at once when the client requests them. (The first case, in which the repository provides the values one at a time, works much like an event queue — see chapter 15 — or channel — see chapter 21.)

Alternately, the repository can keep track only of the last thing deposited. In this case, the repository will work well as long as the server updates the repository often enough that the client is assured of reading a relevant value. If the client doesn't check the repository often enough, though, the client runs the risk of missing some values.

A passive repository can be implemented using a single piece of shared data. For example, the server push may use a setter method, while the client pull uses a corresponding getter method. The data may be simple — a single value or object — or complex, capable of holding many values, like a queue.

Of course, there are both benefits and disadvantages to the use of a passive repository. Advantages include the flexibility to allow the active components to act whenever it may be convenient for them. But problems may arise:

- If the server pushes too much more often than the client pulls — the spy drops off a lot of documents, but the spymaster doesn't claim them — the repository can fill up.
- If server doesn't push often enough for the client — or if the client pulls to frequently for the server — the client may receive out−of−date (stale) information or no information at all. (The spymaster can't pick up documents faster than the spy delivers them.)

The use of a passive repository works best when the client and server need to operate relatively independently, but run at about the same rate.

19.5.2 Active Constraint

An active constraint is the dual of a passive repository. If a passive repository couples a server push server and a client pull client, then an active constraint couples a server push client and a client pull server. Each of these is a passive component — — so they must be joined by a component that takes action: . Note that this component both pulls (from the passive server) and pushes (to the passive client): . Imagine a diner in a fancy restaurant. As soon as the diner puts down his fork, the fork disappears from the table, reappearing at the dishwasher. How does this happen? The active constraint — in this case, the busboy — gets (pulls) the fork from the diner and gives (pushes) it to the dishwasher.

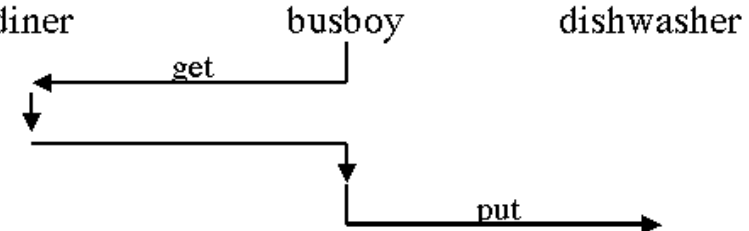


Figure 19.10. An active constraint pulls information from the passive server and supplies it to the passive client.

This process requires no initiation of action on the part of either the client or the server. Instead, each of them goes about their business, responding only when the active constraint explicitly asks for something (or provides something). The intermediate entity — the active constraint — does all of the work to make this transfer happen.

Like server push and client pull, passive repository and active constraint are duals. A server push server can be connected to a client pull client by a passive repository. A client pull server (i.e., a passive server) can be connected to a server push (passive) client by an active constraint. In fact, a passive repository IS the client side of a server push attached to the server side of a client pull. By duals, an active constraint should be the server side of a server push and the client side of client pull — and it is!

Chapter Summary

- A service is something provided by one entity to another. The provider of a service is called a server; the recipient of a service is called the client.
- An entity is a server or a client *with respect to a particular service*. Services can be layered or chained.
- Client pull describes the situation in which a client initiates a service request. This is like shopping at a grocery store, with all the attendant advantages and disadvantages.
- Server push describes the situation in which a server initiates the request. This is like subscribing to the fruit−of−the−month club.
- Client pull and server push are an example of duals.
- A server−push server and a client−pull client can be connected using a passive repository.
- A client−pull server and a server−push client can be connected using an active constraint.

Exercises

1. Real−world interactions are often complicated mixes of clients and servers. One way to tell who is (apparently) the server is that the client often pays for a server's services. Consider each of the following interactions and describe who is the client and who the server:
    a. I buy a computer from a store.
    b. I rent a car.
    c. I rent a computer from a store.
    d. I rent a computer from a service that (i) doesn't charge me but (ii) requires that I read ads before using the computer.

Chapter 20

Synchronization

Chapter Overview

- What happens when two entities want to use the same thing at the same time?

Synchronization is an issue that arises when multiple animacies share state. In Java, this means that there are multiple Threads directly or indirectly accessing some field of an object. These Threads may either be explicitly created or automatically generated as, e.g., the user interface Thread in `java.awt.`

When an object accesses state, it does so either to obtain or to set a value. If the access does not change the value, we call it a ***read***. An access that changes the value of some state is called a ***write***. If more than one thread can access a state, we call it ***shared state***. Shared state can lead to problems if there are multiple accesses of the state at the same time and at least one of those accesses is a write. To avoid these problems, we can prevent sharing, we can prevent writing, or we can use specialized mechanisms or protocols to minimize conflict.

Objectives of this Chapter

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

20.1 An Example of Conflict

When I was in high school, we took a class trip to Washington D.C. While we were there, we had a class photograph taken on the Capitol steps. Since there were a lot of us, they used a panning camera. The photographer started off pointing to the left, then scanned across the class until he got to the rightmost edge. The entire process took a minute or two.

The interesting part came when the photograph was developed. One of my classmates appeared in the upper left−hand corner of the picture. He *also* appeared in the upper right−hand corner! Here's how he did it: He started in the left edge of the group. As soon as the camera had moved past him — during the minute or so that the photographer was scanning the group — this classmate ran from one end of the group to the other. By the time that the photographer got to the right edge of the group, he had reached that side and was standing among the students there.

This is a ***synchronization failure***. The problem is that the scanning of the group took time. Between the time that the scan started and the time that the scan completed, the student was able to change his position, so that the camera recorded him in both places. This is called a read−write conflict: the camera “read” a value that was incorrect. (My classmate's new position had already been recorded. A similar problem would arise if he'd run the other way — then neither position would be recorded.)

A second example arises when two writes conflict. Say that our bank account contains $1000. We go to deposit $100. The ATM (automated teller machine) reads our current balance — $1000 — and goes off to calculate the new balance. At the same time, the bank's computer goes to give us our periodic 1% interest. It, too, reads our current balance ($1000) and sets out to compute our new balance from this.

In the meantime, the ATM finishes computing our new balance and stores it in the central accounting ledgers — $1100. Finally, the banks' computer calculates our balance after interest — 101% of $1000 is $1010 — and writes that value to the central ledger. Unfortunately, the result is that after deposit and interest, we have a balance of $1010, not $1110.

These failures occur because there are two things going on at once — a student running and a camera photographing, or two processes computing new balances — that interact in inappropriate ways.

20.2 Synchronization

Synchronization is required when **two or more threads of control** (animacies) **access the same** (piece of) **state and that state changes**. Synchronization prevents one animacy from reading the state while the other might be changing it. In Java, it also ensures that

each read is of an up−to−date version of the state.

Synchronization is only necessary when there can be a write to shared state.

20.3 Java's *synchronized* Declaration

The primary means of ensuring mutual exclusion in Java is through ***synchronized methods***.

20.3.1 Synchronizing Methods

In Java, a method may be declared `synchronized`. In each object at most one synchronized method can run at any one time. We say that a synchronized method ***obtains a lock*** on its containing object before it can execute. Since there is only one lock for each object, this prevents any other synchronized method from running until this method completes: no other method can obtain a lock on the object until this method releases its lock. This one−animacy−running−at−a−time property is called ***mutual exclusion***.

Locking an object only prevents access to other methods or code blocks that also require a lock on the same object. Locking an object does not prevent other (non−synchronized) methods of the object from running, nor does it prevent other use of the object.

20.3.2 Synchronizing Blocks

Java has a second form of synchronized execution. A special ***synchronized statement type*** can be used to provide mutual exclusion on its body. Unlike synchronized methods, the synchronized statement (sometimes called a ***synchronized block***) must explicitly specify the object it locks. The syntax of this statement is:

```
synchronized (objectReference) {
    statements
}
```

Here, *`objectReference`* is some expression whose value is of an object type; the locked object is the expression's value. The *`objectReference`* expression should be one whose value does not change; otherwise, careless coding can easily lead to a failure of mutual exclusion.

20.4 What Synchronization Buys You

Consider the class photograph described above. If the photographer had had synchronization, he would have been able to tell us not to move — and would have been able to enforce it — until after the photograph was done. My classmate never could have appeared in the single picture twice. (Well, at least not without digital enhancement.)

The bank example is similar. Real ATMs lock the account during the transaction, so that the interest figuring process couldn't read the balance until the ATM was done. In this case, the two computations would not overlap and the correct final balance would be reached.

20.5 Safety Rules

Sometimes, a set of data is interdependent. For example, we might have two fields corresponding to a street address and a zip code. Changing an address might involve changing both of these fields. If the zip code is changed without a corresponding change to the street address, the data may be inconsistent or incoherent. Such a set of operations, which must be done as a unit — i.e., either all of the operations are executed or none are — in order to ensure consistency of the data, is called a ***transaction***. The property of "doing all or none" is called ***atomicity***. A system in which all transactions are atomic is ***transaction−safe***.

The following rule suffices to ensure that your system is transaction−safe:

All (potentially changeable) shared data is accessed only through the synchronized methods of a single object; no interdependent piece can be accessed independently.

Note that this means that shared data cannot be returned by these methods for access by other methods. If shared data is to be returned, a (non−shared) copy must be made. Further, if interdependent values are to be returned (i.e., a portion of the shared data is to be used by other methods), all of the relevant values must be returned in a single transaction.

For example, the address and zip code of the previous example should not be returned by two separate method calls if they are to be assumed consistent.

```
public class AddressData {

    private String streetAddress;
    private String zipCode;

    public AddressData(String streetAddress, String zipCode) {
        this.setAddress(streetAddress, zipCode);
        ...
    }

    public synchronized void setAddress(String streetAddress,
                                        String zipCode) {
        // validity checks
           ...
        // set fields
           ...
    }

    public synchronized String getStreetAddress() { // problematic!
        return this.streetAddress;
    }

    public synchronized String getZipCode() { // problematic!
        return this.zipCode;
    }
}
```

If this class definition were used, e.g. for

```
printMailingLabel(address.getStreetAddress(), address.getZipCode());
```

it would in principle be possible to get an inconsistent address. For example, between the calls to `address.getStreetAddress()` and `address.getZipCode()`, it is possible that a call to `address.setAddress` could occur. In this case, `getStreetAddress` would return the old street address, while `getZipCode()` would return the new zip code.

Instead, `getStreetAddress()` and `getZipCode()` should be replaced by a single synchronized method which returns a copy of the fields of the `AddressData` object:

```
public synchronized SimpleAddressData getAddress() {
    return new SimpleAddressData(this.streetAddress, this.zipCode);
}
```

The `SimpleAddressData` class can contain just `public streetAddress` and `zipCode` fields, without accessors. It is being used solely to return the two objects at the same time.

20.6 Deadlock

If you are not careful, it is not too difficult to get into a situation where multiple active objects each prevent the other from running.

Consider two objects which each need to control both the chalk and the eraser in order to write on the blackboard. The first uses the following algorithm:

1. Wait until the chalk is available, then pick it up.
2. Wait until the eraser is available, then pick it up.
3. Write (and erase).
4. Release the eraser.
5. Release the chalk.

The second uses the following algorithm:

1. Wait until the eraser is available, then pick it up.
2. Wait until the chalk is available, then pick it up.
3. Write (and erase).
4. Release the chalk.
5. Release the eraser.

If the two processes time things just right, it could be the case that they each complete their first steps before reaching their second. Now, the first process will be stuck waiting for the eraser (which the second process has), while the second will be stuck waiting for the chalk (which the first has). This situation — in which neither process can do anything, and both are stuck waiting — is called ***deadlock***. (The processes in this case are effectively ***dead***.)

There is an analogous situation that arises when both processes put down the objects they have and pick up the other object (repeatedly). In this situation, although both processes are still alive, neither is making any progress. This is called ***livelock***.

The desirable property of a system that doesn't reach deadlock is ***liveness***. In general, there is a tradeoff between safety and liveness, and a significant part of programming concurrent applications is designing to simultaneously maximize both.

20.7 Obscure Details

This section is not for the faint of heart. While it is true, it is not pretty. Feel free to skip it.

20.7.1 Synchronization and Local Copies of State

In Java, each `Thread` may keep its own copy of shared state. This means that one copy may be inconsistent with another. Using `synchronized` forces a `Thread` to refresh all of its shared state, ensuring that it does not have a stale copy. Thus, even if timing constraints guarantee that only one `Thread` can access the state at a time, it may still be necessary to use `synchronized`. However, in this case the identity of the locked object is irrelevant; any synchronized method or block will do. (An alternate solution to this problem, though not to synchronization in general, is the ***volatile*** keyword on fields.)

20.7.2 Synchronized Blocks and Lock Object References

It is the value returned by the expression (at the time that the lock is obtained), and not the expression itself, that is locked. For example, given the following class definition:

```
class SynchronizationFailure {

    Object foo = new Object();

    void failToSynchronize() {
        synchronized (foo) {
            foo = new Object();
            other statements
        }
    }
}
```

the synchronized block does not provide proper mutual exclusion. Consider a particular `SynchronizationFailure` instance, `popularObject`. If `Jack` and `Jill` both call `popularObject.failToSynchronize()` with appropriate timing, here is what could happen:

1. `Jack`'s call to `failToSynchronize` obtains a lock on `popularObject.foo`'s current value, say object 1.

2. When the line `foo = new Object();` is executed, `popularObject.foo` is assigned a new value, object 2.

3. `Jack`'s call continues to execute *`other statements`*.

4. In the meantime, `Jill` calls `popularObject.failToSynchronize()`. When `Jill`'s call reaches the synchronized block, it attempts to obtain a lock on `popularObject.foo`'s *current* value, object *2*. Although `Jack's` call is still inside the synchronized block, `Jill`'s call is able to enter because it attempts to lock a *different* object from `Jack`'s call.

Note that this failure can arise any time the value of the *objectReference* expression can change, even when it does not change inside the synchronized block. To avoid such failures, the synchronization expression (i.e., the *objectReference* on which the lock is obtained) should generally be an expression whose value does not change.

Chapter Summary

- Conflict can arise when multiple animacies access mutable state. For example, an entity may read an impossible value.

- This kind of conflict can be prevented by limiting state access to single animacy or by making all shared state immutable.

- When shared mutable state is desired, access can be controlled through Java's synchronization mechanisms.

  - ♦ Each object has its own “lock.”

  - ♦ At most one animacy can hold this lock at any time.

  - ♦ A method may be declared `synchronized`. An animacy cannot execute a synchronized method until it holds the lock of the object to which the method belongs.

  - ♦ A block may be declared `synchronized` *on a particular object*. An animacy cannot execute a synchronized method until it holds the lock of the specified object.

- A transaction is a group of operations which must either be completely executed or not executed at all. Partial execution is not legal. A system is transaction−safe if all of its transactions are executed atomically, i.e., partial execution is not possible.

- In general, increasing (transaction−)safety means decreasing liveness, a program's ability to run towards completion.

  - ♦ Transactions that interfere with one another so that all execution stops are called deadlocked.

  - ♦ Sometimes transactions interfere so that execution continues, but no progress can be made towards completion. This is called livelock.

Exercises

Chapter 21

Network Programming

Chapter Overview

- How do entities on one computer communicate with entities on another computer?

Many modern applications involve multiple computers. This chapter introduces Java's primary mechanisms for making such interaction possible: ***communication channels***, or ***streams***, over which information can be transmitted. Transmission over these channels is often mediated by a ***Lector*** — one who reads — and/or a ***Scribe*** — one who writes — on behalf of an entity. Communication may occur across a network, between co−located entities, or with persistent storage resources such as a ***File*** on disk. The stream abstraction gives these diverse kinds of communication a uniform interface.

In this chapter, we present a series of Lector/Scribes, initially relying only on local resources, ultimately establishing and controlling a network connection. We conclude with a discussion of a multi−threaded server and a brief look at the role of a server in a network architecture.

Objectives of this Chapter

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

21.1 A Readable Writeable Channel

Two entities often need to communicate with one another. We have seen how this can be accomplished using direct method invocation. But that kind of communication requires that the calling object know the identity of the method's owner. This is the equivalent of having a face−to−face conversation: You must know to whom you are speaking. In this chapter, we explore a more abstract communication mechanism that uses intermediate objects — called *streams* — to allow one entity to communicate with another indirectly. Stream communication is more like talking on the telephone. The same device can be used to interact with many different people — or entities — without requiring direct contact. Streams similarly provide a uniform interface that can be used to communicate indirectly with a wide variety of objects or resources.

A ***stream*** — or, more properly, a ***stream of values*** — is an abstractly defined resource containing a sequence of values that can be processed, one by one. Streams come in two flavors: ***input streams***, which support the reading of values, and ***output streams***, which support their writing. In other words, an input stream is a stream from which these values can be read, one by one, in order; an output stream is such a resource to which values can be written one by one. In this chapter, we will concentrate on stream−like objects and how they are used.

21.1.1 Tin Can Telephones

One way to think about a stream is to consider the tin can telephones many of us played with as children:

Picture of tin can telephone; pic of Tweedles talking; stream ends marked.

Take two tin cans with one end removed from each. Punch a whole in the center of the intact end of each can. With a long piece of string, thread the two cans so that their flat ends face each other. Tie knots in the ends of the string. Pull the string tight, so that it is stretched between the two cans. Talk into one can; have someone else listen at the other.

Figure 21.1. Tin Can Telephones.

This is a simple device that allows you to put something in one end and allows someone else to retrieve it at the other end. The end into which Tweedledum is speaking is like an output stream. The end to which Tweedledee is listening corresponds to an input stream. Anything Tweedledum writes to the output stream can be heard by Tweedledee (reading) the input stream.

[Footnote: Note, though, that the communications medium itself — the tin can telephone — isn't an input stream *or* an output stream. The medium has two ends, one of which is an input stream, and the other of which is an output stream. In the case of the tin can telephone, these roles might be marked on the two cans: “Listen here” on one can, “Speak here” on the other. In the case of streams, there's less room for confusion. A stream is either an input stream or an output stream, and the two are not interchangeable.]

One nice property of this system is that the tin can telephone doesn't rely on face to face contact between the conversationalists. Using a stream, a Java entity can talk to all different kinds of things without needing to know much about those things. Communication relies on properties of the stream, rather than on properties of the thing at the other end. It also means that the communicators don't have to be directly in contact, as long as each holds one end of the tin can telephone, or stream. For example, Tweedledum can use the same kind of device to talk to the Jabberwock, even though Tweedledum knows far better than to approach the Jabberwock face to face.

The tin can telephone story so far works well when Tweedledum wants to communicate something to Tweedledee. But what happens when Tweedledee has something to say as well? We can accomplish this using the same approach. But in order to have simultaneous two−way communication, it is useful to have *two* tin can telephones. Then Tweedledum listens to one and talks into the other; Tweedledee listens to the one Tweedledum talks into, and talks into the one Tweedledum is listening to.

Picture of Tweedles talking on *two* tin can telephones.
Also the view from Tweedledee's perspective.

Figure 21.2. Two−way tin can communication.

In the same way, Java streams often come in pairs. One — the −dum to −dee route — is an output stream for −dum and an input stream for −dee. The other — −dee to −dum —

is output stream for −dee and an input stream for −dum. If you are standing at one end — like Tweedledee — you are holding one input stream (from which you can read) and one output stream (to which you can write).

21.1.2 Streams

In this book so far, we have talked about cs101.io.Console, a particular concrete resource that behaves like one end of the two−stream configuration. Console behaves like an input stream — through its *readln* method — and also like a separate output stream — through its *println(String)* method. The Channels used in interlude 1 are another example of stream−like behavior, though the methods provided for reading and writing are somewhat different from Console's. Like those channels, many kinds of Java streams are used to connect two entities.

Streams can also be used to interact with things that may be more outside of your program, such as a File — information stored on your disk — or a network connection. Each different kind of stream−like resource has a slightly different set of (read and write) methods, and each has a very different kind of behavior at the “other” end of the resource. Writing to the Console makes the information appear on your Java console, i.e., on your screen. Writing to a file stores the information away on your disk for later retrieval. Writing to a network connection sends the information to another computer. But from the read/write end — from Tweedledee's perspective — the stream makes these resources seem fundamentally similar.

A stream is general way to think about each of these connections. Individual connections implement different methods to actually make stream communication possible. No matter the differences among them, these methods are generally used in stereotypical ways. The ways that ***reading*** — extracting information from an input stream — and ***writing*** — depositing information in an output stream — are used within a program is the topic of the first part of this chapter. Even the general properties of read and write operations are shared by most of these stream−like connections.

In the first part of this chapter, we will simply use the notation **`inStream.read()`** and **`outStream.write(message)`** whenever we are accessing a *read* or *write* method. All of the code in that portion of the chapter will work if the text **`inStream.read()`** is replaced with the text **`cs101.io.Console.readln()`** and the text **`outStream.write(message)`** is replaced with the text **`cs101.io.Console.println(stringMessage)`**. We use the more generic notation to indicate that other substitutions are equally possible. In the latter parts of this chapter, we will introduce other kinds of streams and talk about other code that might be used to replace **`inStream.read()`** and **`outStream.write(message)`**.

21.2 Using a Channel

This section develops a very general set of classes capable of reading from and writing to a general read/write resource. It is tempting to call the thing that does the reading a ***Reader*** and the thing that does the writing a ***Writer***. However, Java has reserved those names for other classes, described below. Instead, we will call the thing that does the reading a ***Lector*** (meaning "one who reads") and the thing that does the writing a ***Scribe*** ("one who writes"). We will define the interfaces for these classes first.

Although the details will vary from application to application and will depend in some part on the kind of resource from which you are reading or to which you are writing, the general pattern of interaction with a read/write resource is similar. In this section, we will develop a fairly general class whose instances are capable of reading from and writing to resources of this sort. In the succeeding sections, we will modify that class to tailor it to the kind of read/write resource represented by a network connection.

21.2.1 Streams for Writing

Let's say that you have a thing, say, *message*. You want to write *message* to your output stream, *outStream*. Accomplishing this is as easy as it sounds:

```
outStream.write(message);
```

Let's look more carefully at what is going on here. Consider the write end of a channel. This is a stream that implements a `write` method, such as Console.println(String). To use this *outStream.write(Object)* method, all that we need to do is invoke it with the appropriate Object.

The write method of a resource like this one is just a server push client. It accepts information when the server writes it. To use such a write method, you must build code that acts as a server push server towards the write method. It must explicitly take action. That is exactly what happens when we invoke

```
outStream.write(message);
```

21.2.1.1 Flushing Out the Stream

One detail is worth noting. Remember that the output stream is one end of something that has another end (like the tin can telephone). When an output stream's write method is invoked, it causes the thing written — the message — to pass into the stream. It does not, however necessarily cause this thing to be available at the other end of the stream. This is like speaking into a tin can telephone with a noticeable delay between the ends. It is quite possible that the message may be "inside" the stream. It will eventually appear at the other end, but not necessarily when the writer expects it to do so.

Picture of balls stuck inside a pipe.
Figure 21.3. When you write something to an output stream,
it can get stuck "inside" the stream.
A call to the stream's *flush* method will push these objects through.

If it is really important that the information you wrote to the stream not get stuck inside, you can use a special method, called `flush()`, to push the information through. When an output stream's `flush()` method is invoked, anything that's been written gets pushed along through the stream, so that it appears at the other end. If there are multiple things that have been written, they appear at the other end one by one, in the correct order; `flush()` doesn't change anything, it just gets things moving along.

Why might something get stuck inside a stream? Imagine that you have a carton that can hold twelve eggs. You go to the henhouse and pick up an egg. (You "write" the egg to the carton.) Back in the house, the cook is waiting for eggs to make breakfast. But it is silly for you to go back to the house with just one egg if the cook can't start until s/he has enough eggs for breakfast. So you pick up another egg and write it to the carton. You keep going until you have a full carton of eggs.

Streams can work the same way. They can wait until there is a group of information to be sent, then send the whole collection at once. Just as it saves you time to collect a carton full, it can make more efficient communication to wait for a full "packet" of information.

When you invoke a `flush()` method, that causes the information to be sent, regardless of how much is waiting. In the egg collecting example, it would make you go to the house even if you'd only collected two eggs so far. Then you'd have to go back to the henhouse and collect some more. Unnecessary `flush()`es are wasteful, just as in this example.

On the other hand, a judicious `flush()` every now and then can be beneficial. What if you were determined only to return to the house when you had a full dozen eggs? But say that today the hens laid only eight eggs. You might stay in the henhouse until tomorrow rather than return with a partially full carton. In this case, a `flush()` would be just the right thing: It would get the eight eggs you had collected where they needed to go, rather than waiting for the next four eggs (that might never come).

21.2.1.2 A Scribe Example

So far, we have seen how writing to an output stream works. We can encapsulate this knowledge inside a method that takes an object and sends it out over the output stream. The interface for an object supplying this behavior might read:

```
public interface Scribe {
    public void send(MessageType m);
}
```

[Footnote: Note that we are being deliberately cagey about the type of object that can be written (or read). This is because that depends on the specific kind of stream that you're dealing with. Nothing in this section relies on the specific kind of stream or type of message.]

An example implementation of this method (to be encapsulated in an appropriate class) might be:

```
public void send(String m) {
    Console.println(m);
}
```

In other words, a Scribe is an object that keeps track of its output stream and, on (send) request, writes the object to be sent to the stream.

A GenericScribeImpl class implementing this interface would need an output stream. It could then simply use that stream's write method on demand. If it is important that our writing not be delayed, we might add a `flush()` invocation to the send method as well.

```
public class GenericScribeImpl implements Scribe {

    private OutputStreamType outStream;

    public GenericScribeImpl(OutputStreamType outStream) {
        this.outStream = outStream;
    }

    public void send(MessageType m) {
        outStream.write(m);
        outStream.flush();    // (maybe)
    }
}
```

Instances of this Scribe object are suitable for use in event−driven programs. For example, whenever something happens that needs to be communicated, the Scribe's send method could be invoked. It would then write the relevant communication to its output stream.

For example, if we have a TextField and a Scribe:

```
TextField textField = new TextField();
Scribe scribe = new GenericScribeImpl();
```

we might connect them by having the Scribe write out the text in the TextField each time the return key is hit. (Recall that hitting the return key in a TextField triggers an ActionEvent).

This can be accomplished using an actionPerformed method that says:

```
public void actionPerformed(ActionEvent ae) {
    this.scribe.send(this.textField.getText());
}
```

[Footnote: We've omitted a few details from this example. First, the actionPerformed method is embedded in a ScribeListener class whose full definition is:

```
public class ScribeListener implements ActionListener {

    private TextField textField;
    private Scribe scribe;

    public ScribeListener(Scribe scribe, TextField textField) {
        this.scribe = scribe;
        this.textField = textField;
    }

    public void actionPerformed(ActionEvent ae) {
        this.scribe.send(this.textField.getText());
    }
}
```

An instance of this ScribeListener class is then used to connect the TextField with the appropriate Scribe.

```
textField.addActionListener(new ScribeListener(scribe, textField));
```

]

21.2.2 Streams for Reading

Writing to an output stream is fairly straightforward. Reading from an input stream is somewhat more complicated. To help us read from an input stream, we will define a class called Lector: “one who reads”.

The innermost portion of the Lector says something parallel to the Scribe. We certainly want to invoke the stream's *read* method:

```
inStream.read()
```

Immediately, we are faced with the first complication. What should the Lector do with the message read from its input stream? There are many possibilities, depending on what you want your Lector to do. For example, the Lector could just let the user know that it has read the message (through the Java console):

```
Console.println("Lector:  just read "
                 + inStream.read().toString());
```

This line of code reads the message from the input stream, finds its printable equivalent using *toString*, and then prints this version to the Java console. It is one example of a thing that we might want a Lector to do over and over again. We will return to this issue and see more complex solutions below.

The second difference between reading and writing is that the Lector must be an active object. The Scribe is automatically invoked whenever an object is available to be written. But the Lector must check to see for itself whether an object is available for reading. The input stream is passive.

[Footnote: So is the output stream. But the Scribe is activated by the thing that asks it to send.]

The Lector must invoke the input stream's *read* method by itself. This means that an instruction follower has to come from the Lector itself. Not only that, but the instruction follower of the Lector may wind up spending a lot of time waiting for something to become ready to read. When there is no such value, the read request doesn't return. The instruction follower that executed it is simply stuck waiting. This is because reading is a blocking operation.

21.2.2.1 Reading and Blocking

The Lector invokes the input stream's *read* method — asking for the next value — whether or not there's a value ready to be read. It is this ready−or−not condition that poses the real issue. When there is no value to return, the Lector may get stuck waiting for one.

The read operation on almost any kind of stream is called a ***blocking read***. This means that it will not return until the appropriate information becomes available. For example, if you type something on the Java console, ending with the return key, Console's *readln* method will return this String. If you invoke `Console.readln()` again, it will return the next return−key−terminated String that you type. But what if you haven't (yet) typed another return−key−terminated String? In this case, the *readln* method will not return. The method invocation continues until an appropriate String becomes available; the Console's *readln* method waits for a carriage return. This waiting — for the necessary information — is called blocking.

Because stream reading methods almost always are blocking methods, they generally need to be invoked by a dedicated instruction follower, i.e., one that can sit around and wait until the *read* invocation can complete. The blocking *read* method itself is essentially a client pull server: it provides the information on request. To interact with a blocking *read* method, you must write a client pull client: active code that invokes the *read* method on a regular basis.

The fact that the Lector might get stuck waiting for an object to become ready — that the *read* might block — means that the Lector must have its very own dedicated instruction follower whose job is to wait for the *read*. This instruction follower can't be expected to get much of anything else done, because it might spend a long time waiting for the *read* invocation to un−block. We need a dedicated Thread — instruction−follower — who can afford to spend its time waiting. This is like sending one person to stand in line while the others do something. You don't want to tie everyone up standing in line, and if you only have one person, you can't afford to block (wait); you need to hire someone to wait for you.

We can resolve this issue by dedicating an instruction−follower to the read task. This is a job for an animate object.

21.2.2.2 A Lector Example

We are now ready to write the Lector class. The Lector, like the Scribe, keeps track of a stream. But instances of this class, unlike those of Scribe, are animate objects, each with its own AnimatorThread. A Lector can afford to block each time it calls its input stream's *read* method, because it has a dedicated instruction follower. If the instruction follower's invocation of *read* blocks, it is not a problem because this instruction follower is not expected to be doing anything else other than reading from the input stream.

```
public class Lector implements Animate {

    private InputStreamType inStream;
    private AnimatorThread mover;

    public Lector(InputStreamType inStream) {
        this.inStream = inStream;

        this.mover = new AnimatorThread(this);
        this.mover.start();
    }

    public void act() {
        Console.println("Lector:  just read "
                        + this.inStream.read().toString());
    }
}
```

This code shows how a Lector can print the read message to the Console. But this isn't always what we'll want to do when something is read from an input stream. For example, we might want to do a dispatch on case, depending on what the input it reads is. This might involve some giant conditional with **`inStream.read()`** as the `switch` expression. Or we might want to pass the new message around to everyone we know, as in the broadcast server towards the end of this chapter.

This situation should sound vaguely familiar. Something happens: the Lector reads something from the input stream. This is an event. There are many different ways that this event could be handled. In fact, it's not clear that the Lector should do anything itself. Maybe what the Lector should do is to delegate this responsibility to some other object. This could be done using the simple event handling of chapter 15 or the more complex event delegation of chapter 16.

Paralleling chapter 16, let's define an interface for this separate event handler object:

```
public interface LectorListener {
    public void messageRead(MessageType m);
}
```

An example LectorListener class — one whose instances simply print their message to the Java console — might be:

```
public class LectorPrinter {

    public void messageRead(MessageType message) {
        Console.print("Lector:  Just read:  ");
        Console.println(message.toString());
    }
}
```

Now we'll need a way for the LectorListener to register with the Lector. We will assume just one LectorListener per Lector for now, though we could certainly do otherwise (e.g., using a Vector). The modifications are highlighted.

```
public class GenericLector implements Animate {

    private InputStreamType inStream;
    private AnimatorThread mover;

    private LectorListener ll;

    public GenericLector(InputStreamType inStream) {
        this.inStream = inStream;
        this.mover = new AnimatorThread(this);
        this.mover.start();
    }

    public void addLectorListener(LectorListener ll) {
        this.ll = ll;
    }

    public void act() {
        this.ll.messageRead( this.inStream.read() );
    }
}
```

21.2.3 Encapsulating Communications

We have seen how to write the code for a generic Scribe, a class that manages writing to an output stream. We have also seen how to write a generic Lector that actively reads from an input stream. Often, it is useful to package these two functions together. In the single resulting class, we consolidate all management of communications with a single remote entity. This object may add functionality. It may, for example, do some packing or unpacking for us (if we don't want to and receive objects in the same form that we use them within our program). It may do other bookkeeping, for example recording what information comes in or timestamping it. Such a communications manager might also establish the streams initially, handle exceptions, and otherwise provide a single point of contact for the rest of the entities with which it interacts directly. From within its local

community, this entity provides an interface to the remote entity.

Picture of local communications manager

Figure 21.4. A local communications manager.

These two classes can be combined into a single class:

```
public class LectorScribe implements Scribe, Animate {

    private OutputStreamType outStream;
    private InputStreamType inStream;
    private AnimatorThread mover;
    private LectorListener ll;

    public LectorScribe(OutputStreamType outStream,
                        InputStreamType inStream) {
        this.outStream = outStream;
        this.inStream = inStream;
        this.mover = new AnimatorThread(this);
        this.mover.start();
    }

    public void act() {
        this.ll.messageRead(this.inStream.read());
    }

    public void send(MessageType m) {
        this.out.write(m);
        this.out.flush();    // (maybe)
    }

    public void addLectorListener(LectorListener ll) {
        this.ll = ll;
    }
}
```

Note that this class will often have (at least) two instruction−followers active in it: the AnimatorThread named by this.mover and whatever Thread invokes this object's send

method (from outside this class).

21.3 Real Streams

So far, we have been discussing input streams and output streams as hypothetical idealized objects. In Java, there are a series of classes that actually implement this stream behavior. In this section ,we will look at the Java classes that implement stream behavior. All of the classes described in this section are defined in the package java.io unless otherwise specified. Further information on many of these classes are included in the Java IO Quick Reference appendix.

21.3.1 Abstract Stream Classes

Java actually has four abstract classes that implement stream behavior: two input stream types, from which you can read, and two output stream types, from which you can write. The input stream classes are called InputStream and Reader. The output stream classes are OutputStream and Writer. In this chapter, we use the term stream to refer generically to all four of these classes. Each of these classes is abstract, meaning that any instance of that class is actually an instance of some subclass. They are all defined in the package java.io.

A stream is a resource containing a sequence of values. The values in the resource underlying an InputStream or an OutputStream are stored as bytes, i.e., eight bit pieces of data. The values in the resource underlying a Reader or Writer are stored as chars, i.e., sixteen bit data. Certain contexts produce byte streams, while others produce char streams. You do not need to worry about the differences, but you do need to keep track of which one you have.

Every stream has a `public void close()` method. This method frees up the underlying resources that have been used to create this stream. When your program is done with a stream, it should call that stream's close() method. When your program shuts down, any open resources will be closed automatically; however, it is good practice to close your streams as soon as you are done with them.

[Footnote: Although Java includes automatic garbage collection — it will throw away your stream object if nothing in your program can possibly access it any more — Java does not necessarily release the underlying system resource (i.e., the actual connection to a file or whatever else your stream is connected to) at that time.]

`InputStream`, `Reader` and their extensions support a variety of methods for reading. `InputStream`'s *read* method returns a `byte`, while `Reader`'s returns a `char`. `OutputStream`, `Writer` and their extensions support methods for writing. The write method of `OutputStream` takes a `byte` as its argument.

[Footnote: Actually, OutputStream's write method takes an `int`, but it only writes the low order byte of that `int` to the stream.]

The *write* method of Writer takes either a `char` or an `int` or a String.

Each of the abstract stream classes has several subclasses that provide additional behavior. For example, some of these classes provide a wider range of methods, such as *public Object readObject()* and *public void println(String)*. You will often find it more useful to use one of these extended classes. Those classes are discussed in the next sections; their details are summarized in the Java IO Quick Reference appendix.

Many stream methods also potentially throw an exception. The most common exception to be thrown by a stream method is `IOException`.For example, when you go to read from a stream, if the underlying resource has somehow been corrupted, the *read* method may `throw IOException`. When using a stream method, you will often need to `catch` this exception. `IOException` also has several more specific subclasses, each applicable to a particular failure condition.

21.3.2 Decorator Streams

Java uses a technique called decoration to add features to streams. For example, suppose that you have an InputStream but have decided that you'd really rather have a Reader. Java has a class called InputStreamReader that is a special kind of Reader. Specifically, InputStreamReader's constructor takes an InputStream as an argument. The resulting InputStreamReader uses the same underlying stream resource as the InputStream argument, but the InputStreamReader is a Reader, not an InputStream:

Decoration

Suppose you have an InputStream called *in*, and execute

```
Reader reader = new InputStreamReader(in);
```

Now **`reader.read()`** returns the first `char` in the underlying input stream. The streams named *in* and *reader* use the same underlying input stream!

This pattern — adding features by constructing a more sophisticated object around a simpler one — is called ***decoration***. Java streams make extensive use of decoration to

add features. For example, you can now treat *reader* as you would any Reader, decorating it further using the appropriate constructors.

[Footnote: There is, however, no way to make an InputStream from a Reader (or an OutputStream from a Writer).]

Some of the decorations that you might wish to apply to your stream include:

Buffering. This reads a larger group of data from the stream into some hidden storage, and then reads from that storage on demand. This is particularly useful when you are reading from a file or a network connection. Buffering is provided by the BufferedInputStream and BufferedReader classes. BufferedReader also has a particularly useful *readLine* method that returns a whole String, up to but excluding the terminating newline.

Data. DataInputStream is a class whose instances provide a variety of *read* methods that allow you to read Java primitive data. These include *readInt*, *readBoolean*, etc. Note, however, that there is no corresponding DataReader class.

Objects. An ObjectInputStream is very much like a DataInputStream with the addition of a method for reading whole Java Objects: *readObject*. However, only objects that implement the ***Serializable*** interface may be read from an ObjectInputStream.

There are similar decorations on the output side. An OutputStream can be used to create a Writer using OutputStreamWriter's constructor: `new OutputStreamWriter(`*`yourOutputStream`*` )` On the output side, buffering also enhances efficiency, especially when writing to a file or network connection. The BufferedWriter also has a newLine() method. There are also Data and Object OutputStream classes. Only Serializable objects can be written to an ObjectOutputStream or ObjectWriter.

Finally, there are a pair of classes called PrintStream and PrintWriter.

[Footnote: You should use PrintWriter, rather than PrintStream, if you want to create an instance of this kind of output stream. PrintStream exists only for compatibility with certain objects already built in to Java.]

These output stream classes have the special advantage that none of their methods throws IOException. Their methods are called *print* and *println*, rather than *write*, to indicate their non−exception−throwing status. There are *print* and *println* methods for essentially every type of Java primitive. Using an Object's *toString* method, *print* and *println* can also print any kind of Java Object. This makes these output stream types very useful for writing messages, e.g. to the Console.

There are several other decorator stream types defined in the Java.io package. Many of

those are designed for special purposes. A few are documented in the Java IO Quick Reference appendix of this book.

21.3.3 Stream Sources

Now that you know how to manipulate streams, you may be wondering where you can find one. Streams come from a variety of different sources, depending on the resources that they connect.

For example, there are a series of streams that communicate with Files. These are called FileInputStream, FileOutputStream, FileReader, and FileWriter. Their constructors take the name of the file to read from or write to. These streams allow information to be read from or written to persistent storage, such as a disk. Since disk interactions are relatively slow, it is common to combine several disk access operations using the appropriate kind of buffered stream.

Another source of streams is pipes. A PipedInputStream (or OutputStream, Reader, or Writer) can be used to communicate between two Java objects. To do this, you must create a matched pair (PipedInputStream and PipedOutputStream or PipedReader and PipedWriter), then use the connect() method of one piped stream to join it to its mate.

We will look more closely at networked streams — streams that communicate between two computers — below. There are also streams that read from or write to arrays or Strings.

There are two additional streams with which you are already familiar, though you do not know it. These are the streams called the ***standard input*** and ***standard output***. They are the streams that connect to the Java console. So far, you have used these through the cs101.io.Console class. In fact, there are two streams corresponding to the methods of Console.

Both `System.in` and `System.out` are static fields of the class `java.lang.System`. System.in is the standard input (or "stdin") stream, while System.out is the standard output ("stdout"). There a third stream, System.err, the standard error stream ("stderr"), that also writes to the Java console by default. System.err is intended for error messages, while System.out is intended to output to the user.

The type of System.in is simply InputStream. The type of System.out, however, is `java.io.PrintStream`. A PrintStream supports textual output of most Java primitive types as well as objects. It also avoids most of the otherwise-ubiquitous `IOExceptions`.

21.3.4 Decoration in Action

As we have seen, the four abstract IO classes lack some basic useful features and methods. Frequently, you would really rather be using one of their non−abstract subclasses. For example, one very common reader is `BufferedReader`. Instances of the `BufferedReader` class support useful methods such as:

```
public int read() throws IOException;
public String readLine() throws IOException;
```

The first of these reads a single character from the input; the second reads an entire line of input. For a more complete list of `BufferedReader` methods, see the Java IO Quick Reference appendix.

To create a `BufferedReader`, you first need to have a `Reader`. Java doesn't come with any predefined `Readers`, but it does come with a built−in `InputStream`: `System.in`, which reads from the Java console. The following (extremely useful) code assigns the name `myIn` to a `BufferedReader` that gets its input from `System.in`:

```
BufferedReader myIn =
      new BufferedReader(new InputStreamReader(System.in));
try {
    System.out.println("I just read this line:  "
                       + myIn.readLine());
} catch (IOException e) {
    System.err.println("Oops, I couldn't read a line!");
}
```

In general, you can cascade the feature types, i.e., take any arbitrary stream and make another (more featureful) stream out of it. You begin with a particular stream, either based on an external (non−stream) structure or built in. Below, we will focus on streams created from a network interface.

21.4 Network Streams: An Example

In this final section, we will revisit the LectorScribe class that we defined above. Using what we have learned about actual Java streams, we will embellish that class so that it can be used to communicate with other computers running over the network. To do this, we will need to add the machinery of network communications: sockets. In Java, sockets and other network communication classes are implemented in a package called java.net. These classes are also covered in the Java IO Quick Reference appendix of this book.

21.4.1 Starting from Streams

The following code reproduces the LectorScribe class, above, with a few minor modifications. First, we have used actual java.io stream classes (in this case, Reader and Writer) as the stream types. We have also specified String as our *MessageType* and added some error messages when IOExceptions are caught.

```
public class LectorScribe implements Scribe, Animate {

    private Writer out;
    private Reader in;
    private AnimatorThread mover;
    private LectorListener ll;

    public LectorScribe(Writer out, Reader in) {
        this.out = out;
        this.in = in;
        this.mover = new AnimatorThread(this);
        this.mover.start();
    }

    public void act() {
        try {
            this.ll.messageRead(this.in.read());
        } catch (IOException e) {
            System.err.println("Oops, I couldn't read a line!");
        }
    }

    public void send(String m) {
        try {
            this.out.write(m);
            this.out.flush();    // (maybe)
        } catch (IOException e) {
            System.err.println("Oops, I couldn't write a line!");
        }
    }

    public void addLectorListener(LectorListener ll) {
        this.ll = ll;
    }
}
```

Recall the logic of this code: A LectorScribe is responsible for reading from and writing to its streams. This involves continually monitoring the input with an active Thread (in the *act* method) as well as being responsive to requests to write to the output (when *send(String)* is called from another Thread).

21.4.2 Decorating Streams

But what if you are given an InputStream and an OutputStream rather than a Reader and a Writer? In this case, we might add another constructor to this class, one which decorates these byte streams with their char equivalents. Only the additional constructor is reproduced here.

```
public LectorScribe(OutputStream out, InputStream in) {
    this(new OutputStreamWriter(out),
         new InputStreamReader(in));
}
```

Recall that a this() constructor invokes another constructor of the same class. Invoking new LectorScribe(System.in, System.out) results in the invocation of:

```
new LectorScribe(new OutputStreamWriter(System.out),
                 new InputStreamReader(System.in))
```

This would create a LectorScribe that writes on demand to the standard output stream and continually reads from the standard input stream.

21.4.3 Sockets and Ports

Where might you get input and output streams in the first place? The answer depends on what these streams are supposed to connect you to. For example, if you are reading from (or writing to) a file, you could use the FileInputStream/FileOutputStream or FileReader/FileWriter class pairs. In this section, we will explore streams that connect you to other computers. Java contains a standard library package called `java.net` that provides most of the infrastructure for making network connections.

Think back to the tin can telephone example. What we really want is a sort of a place on the other computer that we can connect to: someplace to “plug in” the tin can telephone. Computers have a number of such things, called ***ports***, but you won't see them if you look at the back of a computer. Instead, a port is a virtual place to plug in a special kind of connection, called a *socket*.

A ***socket*** is an abstraction of actual network connections and protocols. It contains two streams: one for input, one for output. In other words, it is the virtual equivalent of a two−way pair of tin can telephones.

To establish a socket connection, you need to run a program at each end (i.e., one program on each of the two computers that the socket will connect).

- One of these programs “listens” for connection requests; this is called the server because it is providing the service of enabling socket connections. The server provides this service on a particular port of its machine. That is, the server needs to know which port to be watching to see whether anyone is trying to connect.

- The other program is called the client, and it contacts the server to open a socket connection. The client program needs to specify what machine to contact, typically using the name of that machine, and also what port on that machine to try to connect to.

Remember that the terms client and server are relative to a particular service. In this case, the server is providing the service of listening for socket connections, while the client is making use of that service. Once the socket is in place, though, it looks exactly the same from both ends.

21.4.4 Using a Socket

Using this idea of sockets, we can now read and write across the network. In Java, a socket is implemented by an instance of the class `java.net.Socket`. Suppose that we have one of these Java `Sockets` and want to read from and write to it using a LectorScribe.

We already know how to create a LectorScribe if we are given either a Reader and a Writer, or an InputStream and an OutputStream. A Java `Socket` has a method to access each of its streams: `getInputStream()`, which returns an `InputStream`, and `getOutputStream()`, which returns an `OutputStream`. If we had a socket — one end of a virtual two−way tin can telephone — we could access its input and output streams using these methods. We can accomplish this using yet another LectorScribe constructor:

```
public LectorScribe(Socket sock) throws IOException {
    this(sock.getOutputStream(), sock.getInputStream());
}
```

In creating a LectorScribe for this `Socket`, we simply extract the streams and use them to create a LectorScribe on an output and an input stream. Using the remainder of the LectorScribe code above, we have a simple program that takes a `Socket` as an argument and transmits what it reads and writes over the `Socket` to the user via the Java console. Note, however, that this constructor risks throwing an IOException. This is because the `Socket` might be corrupt and the streams might not be accessible.

A final note on `Sockets`: Like a stream, a `Socket` has a close() method. You should make a point of closing your `Socket` when you're done with it.

21.4.5 Opening a Client−Side Socket

Now we have code to read and write from a `Socket`, we need to figure out where to get a `Socket` in the first place. As described above, we can get a `Socket` by connecting to a server — a machine that is listening for connection requests — on a particular port. We need to know what machine to connect to, specified by a String corresponding to its hostname, such as `"www-cs101.ai.mit.edu"`. We also need to know on what port the server is listening for our connection. The port is specified by an integer. By convention, ports numbered below 1024 are reserved for “standard” protocols. Otherwise, you have fairly free choice of ports.

A `java.net.Socket` is created by calling its constructor with a `String` corresponding to the hostname of the machine you want to connect to and an `int` representing the port on that machine where something is listening for connections. So, if we had this information, we could use the following LectorScribe constructor:

```
public LectorScribe(String hostname, int port) throws IOException {
    this(new Socket(hostname, port));
}
```

[Footnote: Note that the constructor for `Socket()` may throw IOException.]

This would enable us to say, e.g.,

```
new LectorScribe("www-cs101.ai.mit.edu", 8080)
```

If we put this expression into our public static void main method, running this program would create a program that connects the user to the machine www−cs101.ai.mit.edu on port 8080. Anything the user types would be sent to that port on that machine, and anything that www−cs101.ai.mit.edu writes to port 8080 would be printed on the Java console. This is the complete program!

21.4.6 Opening a Single Server−Side Socket

Of course, to make the client side of this program work, something has to be listening on the appropriate port of the appropriate machine. What code should we run on www−cs101.ai.mit.edu to listen on port 8080?

The port listener code requires another class from the package java.net. This one is somewhat misleadingly named ServerSocket. A Java program uses a `java.net.ServerSocket` to listen for connections. To create a ServerSocket, you need to specify what port to listen on. Remember that this is the local port — the port on the machine this code is running on — and you are not making any connections, just waiting for someone else to contact you. (If someone throws you a pair of tin cans, you

should catch them and use them to communicate.

The port number on which you listen is arbitrary, but it must match the port number on which the client will try to connect. (The client should also use the hostname of the computer on which this ServerSocket is running.) Remember that the port number should be at least 1024.

We will need to add two constructors to LectorScribe. The first simply creates the ServerSocket and invokes the LectorScribe constructor that takes a ServerSocket as an argument:

```
public LectorScribe(int port) throws IOException {
    this(new ServerSocket(port));
}
```

The action is really in this second constructor. This constructor says "listen on your port." The method

```
public void Socket ServerSocket.accept() throws IOException;
```

is a blocking method that returns a `Socket` when a connection has been made:

A ServerSocket's accept() method returns a Socket. Specifically, it waits until some program tries to connect to that port, then returns its own side of that connection.

```
public LectorScribe(ServerSocket serv) throws IOException {
    this(serv.accept());
}
```

This complete LectorScribe is now ready to run on both sides of the network. By having one main program — on a computer named *yourComputerName* — run:

```
new LectorScribe(4321)
```

and the other run:

```
new LectorScribe(yourComputerName, 4321)
```

you can create a simple two−way chat program. The number 4321 is, of course, an arbitrary choice, but both programs must use the same number.

The complete LectorScribe code is included in the code supplement (as LectorScribe.html).

21.4.7 A Multi-Connection Server

The accept() method, like an input stream's *read* method, blocks until there is a connection ready to accept. So, like a **`read()`**, *accept()* — and this method — may wait for a very long time before returning. This means that it may be useful to have the *accept()* invocation run in its own Thread. We can write a variant on the LectorScribe by separating the connection listening from the rest of the program.

In fact, we may want to go further. A single application can have several connections active at once. There is no problem with having multiple connections running over the same port. A port is simply a place where a ServerSocket can be listening for connection requests. For these reasons, it is common to write a more sophisticated kind of server than a simple LectorScribe.

Essentially, the LectorScribe that we have seen so far is run on a Socket, not on a ServerSocket. An additional class is used solely to listen on the Socket. This class needs to have its own instruction follower, so it is an animate object. When it accepts a connection — yielding a Socket — it simply creates a LectorScribe on that Socket.

```
public class MultiServer implements Animate {

    private ServerSocket serv;
    private AnimatorThread mover;

    public MultiServer(int port) throws IOException {
        this.serv = new ServerSocket(port);
        this.mover = new AnimatorThread(this);
        this.mover.start();
    }

    public void act() {
        try {
            new LectorScribe(this.serv.accept());
        } catch (IOException e) {
            System.err.println("Failed to establish a connection!");
        }
    }
}
```

21.4.8 Server Bottlenecks

The server architecture that we have just described puts one computer in the middle of a network. This is sometimes called a ***hub-and-spoke*** architecture, since all connections run through the central server, or hub. There are advantages and disadvantages to this architecture. One of the major potential disadvantages is that the server can be overwhelmed if it receives more traffic than it can handle. In this case, the server has

become a ***bottleneck***, the difficult point where congestion must be relieved. The good news is that in a single−server model, upgrading the server is likely to significantly improve system performance.

Hub−and−spoke architecture is very common in networks. When increased reliability is needed, there are variant architectures that reduce the reliance on a single potential point of failure. The most extreme of these is one in which every computer connects with every other computer (on an as−needed basis). This amounts to a whole lot of LectorScribes talking with each other, without the added MultiServer code. This kind of architecture is called ***peer−to−peer*** communication, because neither of the participants is particularly more important. In that case, one plays the role of the server and the other the client only to establish the socket connection; after that, the two machines are equivalent.

A common variant on the hub and spoke, in which each server is in turn the client of a super−server (which may itself be a client...) makes for more efficient routing. This is called a ***hierarchical architecture***. It is the basis of, for example, computer name lookup (also called ***domain name service***) on the Internet.

Chapter Summary

- An InputStream is a Java abstraction describing an entity from which Things can be read; an OutputStream is an entity to which Things can be written.
- Streams can be used for I/O on the console, files and network connections, as well as certain Java objects like arrays and strings.
- Streams can have features like buffering, filtering, or automatic data formatting. These features can be cascaded using the appropriate stream class's constructor.
- Every Java instantiation has a PrintStream called System.out and an InputStream called System.in.
- ObjectInputStream and ObjectOutputStream are stream types that can be particularly useful for sending objects across the network.
- A Socket is one side of a network connection. It has an InputStream and an OutputStream. You can create a Socket by specifying the hostname and port to which you wish to connect.
- A ServerSocket is something that can accept connection requests on a particular port. You can create a ServerSocket, by specifying which port to listen on. A ServerSocket's accept() method returns a Socket object each time a new connection is made.
- A multithreaded server is an entity that creates a new self−animating object to handle each connection accepted by its ServerSocket.
- Such a server can be a hub for a network, but when it is overloaded, it can also be a communications bottleneck.

Exercises

1. Write code to open a file and read it, one line at a time, printing each line to the standard output.

2. Modify the LectorScribe so that it shuts down gracefully. That is, when a stream throws an exception — e.g., when there is nothing more to read — it should close its streams and its Socket.

3. Modify the MultiServer so that it keeps track of the LectorScribes that it has created. Add something to the act() method of the MultiServer that sends a message over the output stream of each LectorScribe when a new connection is accept()ed. ("Congratulations on your new sibling!")

4. Create a new kind of LectorListener event handler that notifies the MultiServer whenever one of its LectorScribes reads something from its input stream.

5. Combine the answers to the previous two questions so that, when a message is read by one LectorScribe, it is broadcast to all of the LectorScribes' output streams. Bonus: Can you avoid sending the message to the initiating client?

Index

© 2002 Lynn Andrea Stein. Reproduced with permission from *Interactive Programming in Java*.

This chapter is excerpted from a draft of *Interactive Programming In Java*, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as part of Lynn Andrea Stein's *Rethinking CS101 Project* at Franklin W. Olin College (and previously at the Artificial Intelligence Laboratory and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology).

Permission is granted to copy and distribute this material for educational purposes only, provided the following credit line is included: © 2002 Lynn Andrea Stein. In addition, if multiple copies are made, notification of this use may be sent to ipij@mkp.com.

;, 107
+, 158
<=, 164
==, 164
–, 169, 169, 169, 169
+, 79
– operator, 169
abstract, 220, 228, 315
abstract class, 315
abstract method, 138
abstraction, 384
adjectives, 252
agents, 10
AlarmedCounting, 136
alternative, 186, 347
and, 165
animate objects, 249
AnimateObject, 299
applications, 10
argument, 152, 82
arguments, 137
arithmetic operator, 160
arithmetic operators, 158, 159
Arithmetic operators, 171
array, 367, 371, 378
Array Access, 371
array access expression, 371
array construction, 369

Array Construction, 371
array construction expression, 369
array element expression of label type, 370
array element of dial type, 370
array for dispatch, 374
Array Initialization, 371
array literal, 371
array member expressions, 370
array type, 368
Array Type, 371
ArrayOutOfBoundsException, 371, 372
Arrays, 342
ascii, 114
assign, 100
assigning, 99
assignment, 155
Assignment expressions, 171
association, 376
atomicity, 544
autodecrement, 160, 169, 171
autoincrement, 160, 169, 171
AutoIncrement and AutoDecrement, 169
backslash, 114
bandwidth, 12
base case, 430
base type of the array, 368
BasicCounter, 134
batch, 15
binary operators, 158
binary (two–argument) arithmetic operators, 161
bits, 113
bitwise–logical operators, 159
block, 180
blocking read, 559
bodies, 220
body, 135, 183, 219
boolean, 107, 111, 114
boolean operators, 165
boot sequence, 5
booting it up, 5
boots up, 5
bottleneck, 574
bottom–up design, 71
bound, 99
braces, 180

break statement, 361
break statements, 360
Buffering, 566
bugs, 17
byte, 113, 117
call path, 325
callbacks, 424
called, 136, 231
Capitalizer, 77
Capitalizers, 70
cascaded, 378
cascaded – statements, 186
cascaded if, 349, 350
case clause, 366
caseClause, 366
case−sensitive, 100
catastrophic failure, 321
catch, 319, 329, 331
catch clause, 329, 331
catch clause matches, 331
central control loop, 341, 342
char, 114
character, 114, 81
character escape, 114
Character literals, 114
characters, 111
charAt, 81
CheckBoxes, 463
checked exceptions, 334
choice, 8
class, 216
Class, 217
class, 83
class definition, 219
class name, 157
classes, 151, 215, 89
client, 526
clone(), 302
Cloneable, 302
CloneNotSupportedException, 302
coercion, 160
Coercion, 163
coercion, 256
collection, 368, 378
collection objects, 374

Combiner, 70
comments, 139
communication channels, 551
comparators, 164
comparison operators, 159
comparisons, 158
Compound Assignment, 169
compound assignment operators, 169
computer, 7
concatenation, 79
concurrent, 18
condition, 183
conditional, 111, 8
conditional behavior, 343
conditional execution, 183
conditional logic, 351
Conditional statements, 378
Conditionals, 341
conditions, 176
conjunction, 165
Connection, 72
consequent, 183, 344, 347
consequent statement, 344
Console, 96, 98
constant, 154, 255, 356, 359
constant expressions, 354
constant values, 255
constants that are used for their value, 359
"constituting a community of interacting entities.", 11
constructor, 215, 216, 235, 86
constructor expression, 171
constructors, 219
continue, 362
control loop, 11
coordination among instruction–followers, 3
counter, 134
Counting, 134
CountingMonitor, 299
Coupling, 313
cs101 libraries, 98
cs101.util.Console package, 99
data, 221
data repositories, 249
data repository, 257
dead, 546

deadlock, 546
decimal notation, 114
declare, 103, 106
declared, 106, 95
decoration, 565
default, 363, 378
default value, 223
definition, 107, 107, 179
Delayers, 70
design, 16, 249
dial, 116, 94
dial names, 108, 110
disjunction, 165
disk drive, 5
disk server, 526
dispatch, 342
Dispatch, 378
dispatch control, 341
dispatch on case, 354
do loop, 188
document, 139
domain name service, 575
double, 107, 113, 117
double precision, 113
double quote, 114
do/while statement, 188
down−cast, 303
duals, 534
Echo, 311
echo program, 18
else clause, 353
embedded, 378
embedded in an environment, 18
encapsulate, 383
encapsulation, 384
Equality testing, 165
equals(Object), 302
Error, 323, 334
error checking, 323
evaluated, 344
evaluates, 150
evaluating, 176
evaluation for instance−creation expressions, 157
event, 440
event delegation, 464

event handlers, 440
event queue, 449
event–driven programming, 440
Exception, 319, 323, 334
exception handling, 322
exceptional circumstances, 320
Exceptional circumstances, 321
executed, 176
executing, 4
Execution of the – statement, 353
explicit cast expression, 162, 164
Explicit cast expressions, 171
expression, 102, 149, 150, 171
expression conditional, 160
expressions, 89
extend Object, 300
extending, 300
Extension, 313
factories, 252
false, 111, 114
field, 221, 85
field access expression, 156, 171
fields, 151, 156, 213, 215, 219, 252
File, 551
file server, 526
final, 255, 255, 355, 356
final fields, 255
final fields., 356
final parameter, 255, 356
final variable, 255, 356
finally, 331, 331
finally clause, 332
float, 113, 117
floating point, 113
floating point literal, 114
floating point numbers, 117
flow of control, 111, 176
footprint, 141, 146, 232
For Statement, 374
getClass(), 302
getMessage, 323, 334
getter, 259
graphical user interface, 133, 343
grouped into mini–programs and given names, 8
guard expression, 303

(GUI), 133
GUI, 343
helper procedure, 343
“helper” procedures, 341
hierarchical architecture, 575
high level instruction, 7
hub−and−spoke, 574
if, 111
if statement, 183, 344, 353, 378, 378
if/else, 347
if/else statement, 185
IllegalArgumentException, 323
implement, 220
implementation, 17
Implementation, 313
implementations, 220
implementor, 132
include a default case, 363
increment, 134
Incrementable, 252
incremental program design, 13
incremental program development, 17
index, 371
index set, 376
indexOf, 81
infinite loop, 11
inheritance, 297, 298
initial conditions, 252
inline, 259
inner class, 398
Inner classes, 384
input streams, 552
instance−creation expression, 157
instances, 104, 216
instruction−follower, 150
int, 106, 113
integralExpression, 366
interactive, 18
interactive control loop, 18
interface, 132, 133, 135, 146
interface body, 141
interfaces, 131, 89, 96
invoked, 136, 231
invoking, 79
Java, 7

Java console, 98
Java interface, 133, 140
Java operators, 159
Javadoc, 140
keywords, 100
L, 113
l, 113
label, 94
label names, 108, 108
labeled break statement, 361
labeled continue statement, 362
labels, 108
lastIndexOf, 82
latency, 12
layered, 529
Lector, 551, 555
length, 372, 81
length of an array, 371
listeners, 464
literal, 111, 150
literals, 110, 149, 95
livelock, 546
liveness, 15, 546
local state, 85
local variable, 178, 211, 223
local variables, 213
logical conjunction (and), 159
logical disjunction (or), 159
logical negation, 159
logical operations, 158
logical operators, 159
Logical operators, 171
long, 113, 117
loop, 9
loops, 176
lossy, 163
magic numbers, 359
mail server, 526
Math, 262
mechanism for dispatch, 374
member class, 399
members, 219
method, 215, 79, 83
method body, 230
method definition, 229

method invocation, 152, 231
Method invocation, 234
method invocation expression, 171
method overloading, 232
method signature, 135, 146, 327
method signatures, 228
method specification, 137
methods, 135, 152, 216, 219, 252
middle manager, 343
model, 463
MouseListener, 472
MouseMotionListener, 472
multiply evaluating an expression, 352
mutators, 259
mutual exclusion, 543
name, 135, 94
Name Droppers, 70
NameDropper, 77
names, 149, 89
Names, 94
naming, 93
narrowing, 163
negation operator, 166
Network−senders, 70
new, 106
newline, 114
NoClassDefFoundError, 323
nouns, 250
null, 109
NullPointerException, 323
numerical analysis, 114
Object, 300
object, 95, 95
object dispatch, 377
object oriented programming languages, 95
object type, 104
object types, 104
objects, 89, 93, 95
object−type, 104
object−type things, 105
obtains a lock, 543
operands, 158
operating system, 5
operator, 158, 158
Operator expressions, 171

operator−assignment operators, 159
or, 165
organizational design, 11
output streams, 552
overflow, 113
overloading, 142, 146
overriding, 304
Packages, 383
parameter, 136, 206, 82
parameter specification, 135
parameters, 151, 213, 223
parenthetical expression, 166
peer−to−peer, 575
Pig Latin, 70
PigLatin, 78
pixel, 453
polymorphism, 418
ports, 570
postfix, 169
precedence, 167, 168
predicate, 303, 351
prefix, 169
primitive, 93, 94
primitive types, 104, 112
primitive−type things, 105, 110
privacy, 249
private, 256
private constructors, 257
private methods, 256
Procedural abstraction, 342, 383
procedural abstraction, 385, 9
program, 3
program errors, 17
programmer, 6
programming languages, 6
programs, 4
protected, 257, 308
protecting, 256
protocol, 15
prototyping, 17
public, 257
pull, 531
push, 533
read, 134, 541
Reader, 555

reading, 554
real numbers, 117
real time, 15
Rectangle, 220
recursion, 428
recursive case, 431
reference, 94
reference–type names, 119
regression testing, 335
repeated invocation of a method, 352
Repeater, 70, 77
replace, 81
reserved words, 100
Resettable, 141
resource libraries, 249
return, 231, 329
return path, 325
return statement, 191
return type, 135, 137
rule specifications, 135
RuntimeException, 334
scientific notation, 114
scope, 107, 151, 151
scope of a local variable, 223
Scribe, 551, 555
selectors, 259
self–animating objects, 89
semicolon, 107
sequencing, 8
Serializable, 566
server, 526
servers, 10
service, 526
setter, 259
share a reference, 109
shared state, 541
shift operators, 171
short, 103, 113
side effect, 155, 177
side effects, 140
signature, 230, 329
Simple expressions, 171
single quote, 114
single–minded instruction–following, 3
socket, 570

software life cycle, 16
specification, 12
standard input, 567
standard output, 567
state, 221
statement, 107
statements, 89
static, 227, 232
static final, 359
static final fields in an interface, 356
static inner class, 399
step through a collection, 373
stream, 552
stream of values, 552
streams, 551
String, 114, 78, 97
String concatenation, 158
strongly typed language, 95
subclass, 302
substring, 80, 80
super, 305
super();, 310
superclass, 302
switch, 359
switch statement, 341, 354, 366, 378
switches as names, 116
symbolic constant, 354
symbolic constants, 341, 355, 359, 378
Synchronization, 541
synchronization failure, 542
synchronized block, 543
synchronized methods, 543
synchronized statement type, 543
syntax, 100, 150, 157, 97
system, 73
tab character, 114
target, 234, 417
test, 111, 183
test expression, 344, 344, 366
testing, 17
the null character, 223
this, 225, 230, 232, 254, 306, 86
throughput, 12
throw, 319, 327, 327, 329
Throwable, 334

throwing, 324
Throwing an Exception, 326
Throwing an exception, 327
throws, 135, 327, 327, 329
toLowerCase, 80
top level, 219, 222
top−down design, 71
toString(), 302
toUpperCase, 80
traditional objects, 249
transaction, 544
transaction−safe, 544
Transformers, 71
trim, 80
true, 111, 114
try, 331
try block, 329
try body, 331
try/catch, 329
try/catch/finally, 331
try{}catch(){}finally{}, 331
type, 103, 103, 103, 105, 105, 150
type of a method, 137
Type−of−thing Name−of−thing, 106
types, 218, 89, 89
Types, 93
typing, 93
Ubby Dubby, 70
unary minus operator, 161
unary operators, 158
unicode, 114
unlabeled continue statement, 362
up−cast, 303
user, 73
User interface, 133
user interface, 74
users, 132
value, 150
value−type names, 119
variable, 151
Vector, 304
verbs, 251
view, 463
virtual fields, 261
visibility, 308

visibility protectors, 383
void, 192, 229
volatile, 547
web browser, 5
web server, 526, 5
while, 187
white space, 100
widening, 162, 163
write, 541
Writer, 555
writing, 554