

Simple Computer Games

Using Java

Dr. Brian C. Ladd
Dr. Jam Jenkins

Contents

Contents	i
1 Getting Started: What's in a Game?	1
1.1 What's In a Game?	1
1.2 Active and Passive: Rules Followers	6
1.3 Running a Game	9
1.4 Strategies: Winning a Game	13
1.5 What is in a Computer Program?	14
1.6 Summary	19
2 Designing Your First Program	21
2.1 Drawing a Playing Card	21
2.2 How Computers Work	24
2.3 Java	29
2.4 FANG	34
2.5 An Introduction to Sprites	38
2.6 Get the Picture	55
2.7 Summary	58
3 Deciding What Happens: if	65
3.1 A Simplest Game	65
3.2 Computer Program (Game) Design	68
3.3 Sequence	70
3.4 Selection	77
3.5 Finishing NewtonsApple	80
3.6 Summary	81
4 Components: Names, Types, Expressions	85
4.1 Randomness in a Game	85
4.2 One More Sprite: CompositeSprite	90
4.3 Java Components	99
4.4 Examining a Public Interface	111
4.5 Finishing EasyDice	116
4.6 Summary	121
5 Rules: Methods, Parameters, and Design	125
5.1 A Simple Arcade Game: SoloPong	125
5.2 Top-down Design	129
5.3 Delegation: Methods	131
5.4 Expressions Redux	136
5.5 Finishing Up SoloPong	140
5.6 Summary	146

6	Components Meet Rules: Classes	151
6.1	Playing Together	151
6.2	Network Programming with FANG	151
6.3	Abstraction: Defining New Types	154
6.4	Finishing the Game	158
6.5	Summary	166
7	Collections: ArrayLists and Iteration	169
7.1	Flu Pandemic Simulator	170
7.2	Console I/O: The System Object	172
7.3	Iteration	176
7.4	Collections: One and Many	182
7.5	ArrayList is an Object	188
7.6	Finishing the Flu Simulation	189
7.7	Summary	199
8	Multidimensional Data Structures	201
8.1	Rescue Mission	201
8.2	Inheritance	202
8.3	Multidimensional Collections	207
8.4	Animation	215
8.5	Finishing Rescue Mission	218
8.6	Summary	224
9	Scanner and String: Character Input	227
9.1	Designing Hangman	227
9.2	Starting Programs	232
9.3	Different Iteration	237
9.4	String Manipulation	240
9.5	Reading Files	246
9.6	Finishing Hangman	254
9.7	Summary	256
10	Console I/O: Games without FANG	259
10.1	Another Dice Game: Pig	259
10.2	Pure Console I/O	263
10.3	Sorting a Collection	268
10.4	Finishing Pig	274
10.5	Summary	280
11	More Streams: Separating Programs and Data	281
11.1	Outsmarting the Player: 20 Questions	281
11.2	Reading and Writing Files	290
11.3	Data-Driven Programs	292
11.4	Encoding Objects to Read or Write Them	293
11.5	Finishing the Game	299
11.6	Summary	304
12	Lists of Lists and Collision Detection	305
12.1	Designing BlockDrop	305
12.2	Software Engineering: Managing Complexity	306
12.3	When It's Safe to Move: Collision Detection	315
12.4	Finishing BlockDrop	327
12.5	Summary	330

13 String Processing: Interactive Fiction	331
13.1 Back to the Future: Interactive Fiction	331
13.2 Iterative <i>Development</i>	333
13.3 Reading The Data	334
13.4 Incremental Development	352
13.5 Finding a Match	356
13.6 Summary	359
A Java Language Keywords	361
B References	363
C Java Templates	367
D FANG Color Names	369
Index	371

Dr. John B. Smith at UNC Chapel Hill for helping me realize that the random ideas I had about introductory programming texts were aspects of an event-driven introduction to programming. Also for being my academic and thesis advisor.

Lynn Andrea Stein's *Rethinking CS101* project for starting a discussion (in my head) about what expectations in introductory programming books should be.

Kim B. Bruce and the *Java: An eventful approach* gang at Williams College for reviving my interest in an event-driven introduction to computer programming.

Game names, game company names, and video game program names are the property of their respective trademark or copyright holders.

Preface

Motivation

Mining the Past for the Future

How to Read Syntax Templates

Computer languages are artificial languages designed to eliminate as much ambiguity as possible. Because they are designed to be processed by computers, the structure of the language can be formally expressed.

The Extended Backus-Naur Form (EBNF) is a mathematically formal way of expressing the syntax (structure) of a programming language. The notation was developed by John Backus and enhanced by Peter Naur more than forty years ago.

We will use an EBNF notation to describe templates for various language constructs. The notation will use [language=template]<somename> to indicate named *non-terminal* symbols, that is, symbols which stand for other templates defined somewhere else. Symbols outside of the angle brackets, except for those special characters defined here, stand for themselves.

The special symbols are:

- : = Defined as: the symbol to the left of this symbol is defined by the template on the right of the symbol.
- [. . .] Group the elements between into one expression for application of the various repeat and optional markers.
- | Selection between the prior and following expressions. One or the other can appear.
- ? Optional: the item this follows can appear zero or one time.
- + Optional repeat: the item this follows can appear one or more times.
- * Repeat: the item this follows can appear zero or more times.

So, as an example, a floating point number, a number with an optional decimal part, could be written with the rules:

```
<floatingPoint> := -? <digit>+ [ . <digit>+ ]?  
<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This says that a floating point number has at least one digit to the left of the decimal point (thus . 1 is *not* valid according to this template) and if there is a decimal point, there must be at least one digit to the right of the decimal point. The decimal point (and any following digits) are optional. The negative sign is optional and, by this definition, a leading plus sign is not valid.

The point of using these templates is that they are unambiguous and show exactly what is valid when declaring some particular Java language structure.

One note is that we will use simplified templates early in the book, templates which do not include all of the complexities permitted in a given structure. As we make use of the complex features we will repeat the template with the more complex parts. Thus the *last* template presented in the book is the most complete. To aid in studying, the templates presented in each chapter are repeated at the end of the chapter in the summary and the most complete templates are all collected in Appendix C.

Getting Started: What's in a Game?

While some computer scientists are sometimes loathe to hear it, creating a computer program is as much art as it is science. Converting an **idea** for a computer program into a **programmable description** of that idea is a *design task*; in fact, that phase of development is called the *design phase* by developers. Designers of all types have rules that they apply yet there remains an *aesthetic* dimension which cannot be learned from a book; it is learned only through practice.

This is actually a very wonderful thing: computer programming, from scratch to running code, involves both *creative* and *analytic* capabilities. It uses all the parts of the programmer's brain that can be brought to bear.

Learning the aesthetic component of a design skill requires *practice* and evaluation of the results. One of the most important parts of the practice is to play with the elements, trying different combinations and trying for different effects. This book uses computer games as a programming area to teach general computer programming. The preface discusses the approach in detail, but there are two primary reasons for this: computer games encompass all the hard problems in computer science and making computer games will trick you into playing with the computer programs.

Playing the computer games in this book will motivate you to improve them; some improvements are suggested in the text or exercises but you are sure to come up with other, better, changes on your own. To tweak what happens in the computer *game* you will have to tweak what happens in the computer *program*.

This chapter examines the parallels between a game and a computer program, starting by looking for a usable definition of a game. The following chapters begin giving you the tools to build your own game using the Freely Available Networked Game Engine (FANG) and the Java programming language.

1.1 What's In a Game?

Consider for a moment the seemingly simple question: What is a *game*? Do you know what a game is? Everyone I have ever asked that question answers, "Yes," yet almost everyone balks when I follow up with the obvious, "Well, what is it?" Before we can compare games and programs, we need working definitions of both.

Definitions

Before proceeding, we should examine that last statement above: Do we need a definition of a game? In *The Art of Game Design* [Sch08], Jesse Schell spends almost two pages talking about academics, game researchers, demanding detailed game definitions, definitions which do not interest practicing game designers.

The rant stung as the authors are both academics who study and teach with games. As Schell worked through the next dozen pages, he found some merit in the struggle to define games: a definition provides a mental framework for holding the pieces we learn about game design and programming. Having a working definition of a game also makes it easier to exploit parallels between game and program design and implementation; it lets us analogize from one field to the other.

Dictionary Definitions

When getting started with a definition, the dictionary is often a rewarding place to start.

The *Oxford English Dictionary* provides an interesting starting place with “**4. a.** a diversion of the nature of a contest, played according to rules, and displaying in the result the superiority either in skill, strength, or good fortune of the winner or winners.”[OED71] Alternatively, “**8. c.** the apparatus for playing particular games.”[OED89] Each of these definitions provide useful pieces: a game has *rules* and is played with *apparatus* of some sort. If you think about a simple computer game, the game has virtual apparatus of some kind and some sort of rules.

Other dictionaries have the same general definition though the *American Heritage Dictionary* is one non-specialized dictionary providing a mathematical definition: “A model of a competitive situation that identifies interested parties and stipulates rules governing all aspects of the competition, used in game theory to determine the optimal course of action for an interested party” [AHD00]. Again we see rules but these rules apply to the party *playing* the game rather than to the apparatus *in* the game. This definition might be useful when developing *strategies*, ways of playing games, particularly when constructing computer opponents.

The current popularity of game design in general and the study of video games in particular means that there is a considerable specialized literature on games and play. Perhaps their definitions are better suited to our need.

Literature Definitions

Salen and Zimmerman, in *Rule of Play*, relate *play* and *game* by noting that not all languages have two different words for the two concepts. Further, they note that, “Games are a subset of play,” and “Play is a component of games.” [SZ04] Of all the play we do, only some of it is games (the first statement) yet when we use a game, we play it (the second statement). This permits us to look at definitions of play as well as games.

Johan Huizinga, a Dutch anthropologist, published his *Homo Ludens* (“Man the Game Player”) in 1938, offering one of the first and broadest academic definitions of play:

[Play is] a free activity standing quite consciously outside “ordinary” life as being “not serious,” but at the same time absorbing the player intensely and utterly. It is an activity connected with no material interest, and no profit can be gained by it. It proceeds within its own proper boundaries of time and space according to fixed rules and in an orderly manner. It promotes the formation of social groupings, which tend to surround themselves with secrecy and to stress their difference from the common world by disguise or other means. [Hui55]

What does this have that our dictionary definitions lack? It talks about play as being outside of ordinary life, bounded in time and space; he goes on to talk about the “magic circle” which people enter when they begin a game, the magic circle containing the rules which apply while the players are in it. The magic is further reflected in the total absorption of players while the game lacks seriousness or real world consequences.

David Parlett, a game historian, offers a definition of *formal* games in the *The Oxford History of Board Games*:

A formal game has a twofold structure based on *ends* and *means*:

Ends. It is a contest to achieve an objective. (The Greek for game is *agôn*, meaning contest.) Only one of the contenders, be they individuals or teams, can achieve it, since achieving it ends the game. To achieve that object is to win. Hence a formal game, by definition, has a winner; and winning is the “end” of the game in both senses of the word, as termination and as object.

Means. It has an agreed set of equipment and of procedural “rules” by which the equipment is manipulated to produce a winning situation. [Par99]

This definition captures the idea of challenge or competition in a game as well as explicitly mentioning both the game rules and the game equipment.

There are literally dozens of attempts to define game and play in the emerging game studies literature as well as in game design guides, books that are less academic but more focused on helping budding game designers learn the skill of game design.

Jesse Schell, in *The Art of Game Design* examines a number of definitions before settling on our final and shortest definition of play and games:

Play is manipulation that satisfies curiosity.... A game is a problem-solving activity, approached with a playful attitude. [Sch08]

This is of particular resonance for a computer programmer because computer science is, fundamentally, problem solving. All a computer program is is a formal description of how to solve a problem. This definition does not go into detail about the rules and equipment but that is what the rest of Schell's book is about.

Game Studies

Game studies is a fairly young field of academic study having grown up in parallel with the rise of the video game industry. Prior to the 1980s, only a handful of historians and anthropologists studied games and play and their definition. Johan Huizinga, in his 1938 *Homo Ludens* (quoted in the chapter), offered a book length examination of games across cultures and across history.

Several different approaches to the study of games have emerged. Anthropologists remain interested in the meaning of games to the people who play them. Sociologists and economists study how people interact with games as systems. New media scholars study games as artistic constructs, examining how they communicate their different messages. Computer scientists, game designers, and other technologists focus on how games are made and the industry that makes them. In the following descriptions of specific research, notice how interconnected the different approaches are.

Study of the meaning and impact of games on players includes looking at games as learning tools, games as simulations, and how games change their players. The military has been interested in games in the first two senses here since before World War II with the use of flight simulators. As early as the 1940s *analog computers* were used to calculate the outcomes of the settings of flight controls.

Learning with games has begun moving into the mainstream with Gee's *What Videogames Have to Teach Us About Learning and Literacy*[Gee03] being a recent attempt to look at **what** players learn from the games they play and how that experience can be used to reach modern students. The negative impact of games is also a focus of this approach. *Grand Theft Childhood: The Surprising Truth About Violent Video Games*[KO08] by Kutner and Olson is a popular press book surveying this research. This research impacted the games presented in this book in that there is a conscious effort to limit the amount of violence in any of the game designs. This is a personal, designer-specific decision, but the book attempts to create games with playable mechanics without a violent veneer.

Sociologists look at the social constructs surrounding games. Some study massively-multiplayer on-line games (MMOs) and the societies that arise within them. Others look at the "mod communities", the groups that form around different games to modify and extend them. Mia Consalvo's *Cheating: Gaining Advantage in Videogames*[Con07] looks at players who subvert the social contract in multiplayer games. Another interesting approach to examining MMOs is taken by economist Edward Castranova in *Synthetic Worlds: The Business and Culture of Online Games*[?] which looks at on-line games internal economies as models of the real world.

Media studies looks at the message being sent by the game designer. This is closely related to what/how games teach their players. Many in this field, like Ian Bogost (*Persuasive Games*[Bog07]) look at the "accidental" messages contained in some early and commercial video games and look for ways to use the same methods to actively communicate a message.

The book in your hands is a result of the computer science approach to game studies: looking at how games are made exposed the deep computer science problems in video games and that was then brought

Game Studies (contd.)

back to the introductory classroom. Others, including Rudy Rucker in *Software Engineering and Computer Games* have had similar insights.

This approach also includes the massive number of game design books. Chris Crawford's *The Art Of Computer Game Design: Reflections Of A Master Game Designer*[Cra84] is a seminal work in this field. Salen and Zimmerman's *Rules of Play: Game Design Fundamentals*[SZ04] is a more recent, incredibly comprehensive take on game, and in particular, video game design.

A recent, readable overview of the state of video game studies as a whole is James Newman's *Videogames*[New04]. The fledgling field of game studies has only recently begun having its own academic journals and conferences; the continuing popularity of video games indicates that the field will continue to grow apace.

A Working Definition

Looking at these sample definitions, we can extract our own more general and more detailed definition:

A *game* is a collection of components that interact according to a set of rules presenting the player with a meaningful choices to make in determining the outcome of a struggle with the environment or another player.¹

The “meaningful choices” is an attempt to capture the problem solving in Schell's definition; these are also where the absorption of Huizinga's definition emerges. The “outcome” of the game reflects the contest elements in Parlett's definition and implies we will want to have ways of keeping score.

This definition gives us names for the parts of a game, a way to talk about how a game is built. It also gives us names for the parts of a *computer game*.

The remainder of this section explores components and rules as they appear in games. Following sections explore how they apply to computer programs and computer games.

This chapter talks about computer programs but it does not include any; do not use that as an excuse to skip it. It is not that long and the insight you gain from this parallel presentation will help you explore the computer programs presented in all of the following chapters.

Things: Components

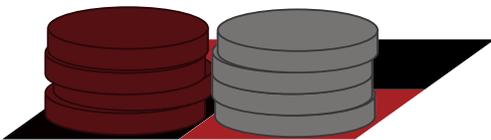


Figure 1.1: Components of Checkers

Components are the *things* used to play a game. A traditional game of checkers, for example, has sixty-four squares in two colors arranged in an 8 × 8 alternating square, a checkerboard, and twenty-four checkers, also in two colors.

There are two flavors of components: *passive* components (like those in checkers) and *active* components which move on their own (due to physics) or add something new to an ongoing game.

¹This definition fails to address the “entertaining” and “amusing” portions of the dictionary definitions; these dimensions are subjective and difficult to quantify. This book uses games to teach software design and development; while some mention will be made of “fun”, the player experience is more properly the domain of game design. Interested students are referred to the references at the end of the book.

Passive Components

Passive components are place holders, used to keep track of the game's progress. The checkers and board illustrated above, the playing cards in bridge or poker, or the peg board in cribbage are all examples of passive components. The players of the game manipulate the passive components according to the rules of the game.

Active Components

Active components are components which “make decisions” about how to contribute to the game. The “decision maker” differs for different components: dice and marbles move according to the laws of physics (interacting with the table, gravity, one another, and friction); collectible cards have their automatic actions enumerated on them or listed in the rules for the game.

Cubic dice, when rolled, bounce and rub on the surface to provide the players with a random number, something which was not there before. Other games' active components include the marbles in Milton Bradley's *Hungry Hungry Hippos*[™] or the creature cards in the various collectible card games (Wizards of the Coast's *Magic: The Gathering*[™] or Nintendo's *Pokémon*[™]).

Actions: Rules

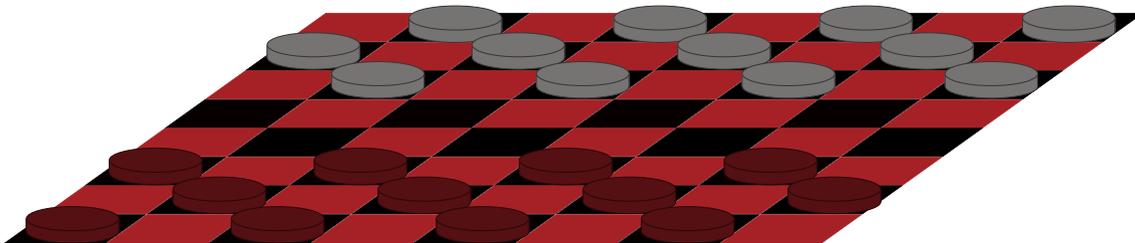


Figure 1.2: Checkers Opening Position

Rules determine how the components of the game interact: the starting configuration of components, legal moves to change the configuration, how the configuration is evaluated for scoring, and how to know when the game ends. The rules also determine the winning player and even if there is a winning player.

Consider traditional checkers again. The rules specify the opening configuration shown in Figure 1.2. Players then alternate moving one checker at a time diagonally onto an adjacent, unoccupied square. Initially, all checkers may only move toward their opponent's rear rank; checker pieces are *promoted* when they reach their opponent's rear rank and as *kings* can move in any diagonal direction. Checkers may *jump* an opposing checker if they are adjacent to it and there is an unoccupied square just past it in a direction that checker could normally move. The game ends when one player loses his last checker or cannot make a legal move; the player unable to move loses the game. The rules prescribe each player's goal within the game as well as how checkers and board squares interact.

A set of rules can be applied to different components without changing the game: chess can be played on a pocket board, with life-sized pieces, or even on a piece of paper by mail. The components differ in scale but not in kind; the same set of rules means the *same* game.

The same is not true of games sharing only components. Both bridge and “Go Fish!” use a deck of 52 cards divided into four suits with thirteen values each. The *relationship* between the components, embodied in the rules, is different. The initial configuration, sets of legal moves, and even each player's goals differ. This is a different game.

Review

(a) You are trying to create your own game to play with two six-sided dice and 2 or more players. The game is made up of rounds. The first round begins by each player rolling the dice and marking down the number

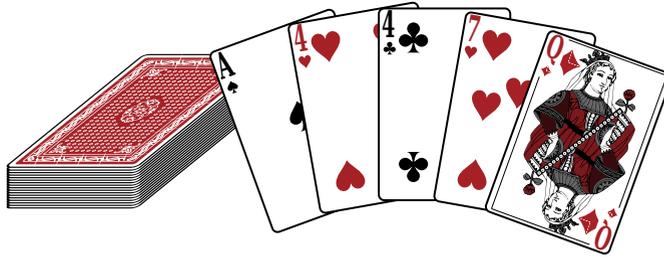


Figure 1.3: Playing Cards

that is the sum of the two dice. This number is the starting score. After everyone has had a chance to roll the dice, the first round ends. During each successive round, the player with the lowest score rolls first and then the player with the next lowest score rolls next, and so on until all players have rolled during the round. After each roll, the sum of the die rolls is added to the previous sum. The first player to reach or exceed 100 wins the game. According to our definition, is this a game? Why or why not?

(b) Chris Crawford, in *The Art of Game Design*[Cra84], differentiates between a *puzzle* and a *game*: a puzzle presents the player with a configuration and rules for changing the configuration but the configuration is fixed and to be *solved*. A game has *interactivity* in that the configuration changes over time with or without intervention of the player. Does our definition capture this difference? Or would our definition admit a puzzle as a game? Is such a difference important?

(c) Can you name three games played primarily with a pair of six-sided dice? Describe the rules of each. How are *turns* different between the games? How are *winning conditions* different between the games? Could any of the rules work with twelve-sided dice instead of six-sided?

1.2 Active and Passive: Rules Followers

The rules provide the user with the choices they make in their struggle to win the game. How are rules followed for active and passive components? Can we reconcile the two different kinds of components?

Who Follows the Rules?

In a turn-based game with only passive components, who follows the rules? As a practical question, how are the rules being interpreted and how are game configurations modified so that only legal configurations are used?

Consider players **A** and **B** playing a game of checkers. As they alternate turns, each follows the rules as part of making their turn. When player **A** makes his move he considers the rules of the checkers; he is, of course, also considering how he is going to win. He follows the rules of checkers in making a legal move and the rules of his strategy by picking his “best” move from those available. After player **A** moves, player **B** takes over, following the rules of checkers and her own strategy for the duration of her turn. Thus **A** and **B** together provide the rules follower for their game.

What if we add a new active component, a *moderator* who mediates the players’ interactions with the passive components. The players limit their interaction to the moderator and the moderator, following the rules of the game, interacts with the remaining components.

The Moderator

Instead of moving components on their own, the checkers players in a game with a moderator take turns submitting their moves to the moderator. The moderator then follows the rules of checkers.

To keep the moderator as simple as possible, we enhance the checkers: each checker indicates all currently legal moves and, when moved, which opposing checkers (if any) are to be removed and whether or not the moving checker should be promoted. The movement rules of checkers are embedded in the checkers themselves; they are followed by the moderator.

The moderator's rules are simple and generic:

```
while current player has at least one legal move
  ask current player to pick a checker
  if checker is not legal
    back to top of loop
  else
    ask player to pick move
    if move is not legal
      back to top of loop
    else
      make move
      change current player

current player loses/other player wins
```

Though the moderator's rules say "checker", if that is changed to "component" then these rules are completely generic. They do not depend on what game is being played: the moderator follows the *game loop*.

The Game Loop

The game loop is at the heart of any moderator proctoring a game; it is also at the heart of any computer game or other interactive computer program you have ever run. At its most general, the game loop is:

```
while (not game over)
  show the game state
  get input from the user
  update game state

indicate winner (or tie)
```

How does that match the loop above? The moderator, an example of a *rule follower*, do not actively show the state of the game (players look at the board) but otherwise follows the above loop fairly directly.

The only thing the moderator knows about checkers is that players alternate making legal moves and that a player loses when they have no more legal moves. This is because we have placed all the complicated movement rules inside the components that are moved. A turn-based game like checkers, one with no active components other than the moderator, can be played with a single rule follower. The moderator asks each player in turn for his action, waiting and doing nothing while they think about the move. The moderator then follows his rules (and those embedded in the checkers) according to the current player's request. Players concentrate on their personal *strategies* or rules for *winning* the game (more on strategies in the next section); the moderator concentrates on enforcing the rules to *play* the game. It is important to note the distinction between the rules of the game (which are inherent in checkers) and the strategy each player employs.

Note that each active component in a game has its own rule follower. In *Hungry Hungry Hippos*, every marble is moving according to the rules of physics *at the same time*. Each hippo waits for a player to tell it to move (by pressing their button) but any player can tell her hippo to move at any moment and the hippo must respond by biting into the arena. The marbles respond to gravity (and being hit by hippos), so the laws of physics determine their motion; each of these components moves independently and has, for our purposes, its own rule follower.

Rule followers are event-driven. That means that they act in response to things that happen. The moderator asks a question and then waits for an answer, continuing with his work only when an answer is given. The hippos are similar. The marbles, too, are waiting for things to happen to them (being hit by another marble, being swallowed by a hippo) but rather than sitting still while waiting, the marbles also move around on

the board. So the two patterns of action we see with the rule followers here are waiting, doing nothing, for some trigger event to set off a sequence of rules or the continuous following of rules. In either case, each independent active component in the game has its own rule follower.

This section defined a game as a collection of components and a collection of rules that determine how the components interact, presenting the player with meaningful choices influencing the outcome of the game. Players (in this section) are outside of the game and interact with the components either one at a time or simultaneously. Components are either active or passive. Active components have rule followers that permit them to act simultaneously and they react to certain events. Central control for a game can either be manifest in a particular component or can be provided by the players collectively. What players choose to do is guided by their strategies, the rules they use to try to win the game.

Retrogaming

Retrogaming, also known as *old-school gaming* or *classic gaming*, is the hobby (or obsession) of playing games designed for a bygone era of gaming hardware. Some play, like one of the authors, because they fondly recall when *Space Invaders* and *PacMan* took over their lives. Some play to experience some of the best crafted games ever designed. Some play to explore specific designers' work. And, finally, some play because retrogames have been introduced on modern game platforms, bringing the Golden Age of arcade games alive for a new generation.

Retrogames are played in three ways, each providing a different take on the original experience: on retro hardware, in emulation, and in ports. Retro hardware is the purist's choice: find original game systems, game or console, restore them to working order, and play your heart out just as the game was designed.

Looking at games from the early 1970s it is important to remember that game machines were *not* general purpose computers that happened to play games. They were custom-built, special-purpose computers that *only* played a specific game. This was true of the quarter-draining machines in the local arcade (a market niche virtually created by *Space Invaders* and *PacMan*) and it was true of the first generation of home game consoles.

From 1972 to 1976 home game systems like Magnivox's *Odyssey* and Atari's *Pong* had one to five games hardwired into them; hardwired in this cast to be taken literally because the circuit boards and custom logic chips really were wired to play only a few different games. In 1976 Fairchild Electronics introduced the **Channel F**, the first home game console with a computer at its heart. Games were bought on cartridges which plugged in and provided the program for the computer to run. It was also the first home console to have enough memory to provide computer-controlled opponents for games.

The following year Atari introduced the **Atari 2600** (also known as the **Video Computer System (VCS)**). The **2600** was a runaway best seller, creating the real market for video game consoles spawning hundreds of game titles from dozens of publishers.

Generations of consoles are referred to by the number of bits the computer processes at one time; more bits means more power. The 8-bit generation included the **VCS**, Mattel's **Intellivision**, and, after the Video Game Crash of 1984, the original **Nintendo Entertainment System (NES)**. The **Super NES** and the Sony **Playstation** (redubbed the **PSone** in 200) were 16-bit winners. While the bit-size wars have ended, the computing and graphics power of consoles has continued to grow in each successive generation.

Retrogamers, longing to play the good old games but unable to find or afford the original hardware, sometimes play the original software on a console *emulator*. An emulator is a program, running on a general purpose computer which "behaves like" the original hardware. The original game software instructs the emulator just as it would the console and the emulator drives the modern hardware to make it play a game like the original. The idea of hardware *virtualization* is very similar to emulation. In virtualization the machine software behaves as if it were a complete computer of the *same type* as the one it is actually running on; emulation is used when the two types of computer are different.

The Multiple Arcade Machine Emulator (MAME) [MAM09] is an emulator that can play retrogames from the 8-bit through modern eras on current PCs. As a legal note, the game software for the original system is, quite probably, under copyright and is not free to download and share.

Retrogaming (contd.)

Retrogames are also available as *ports*. A port is when a computer program written for one type of computer is translated into a program for a different computer; what the program *does* remains the same though how it does it is changed. Games are ported to gain revenue from popular titles. It is also done to make use of games designed for limited hardware: these games are well-suited to the computing power now available in mobile devices. Ports are also popular on Internet distribution channels like the **Xbox Arcade** and the **PlayStation Network**; the titles are inexpensive and appeal to a casual gamer demographic.

The games you will build in this book owe a great deal to the Golden Age (1977-1983, approximately), the 8-bit generation. Tight constraints on graphics and computing, like the linguistic limitations when writing haiku, drove designers to create masterpieces of minimalist beauty. The games here pay homage to the designers of that era but the limitations are used to keep the focus on computer programming. You, the reader, still get to create fun, playable games while you learn the fundamentals of computer programming.

Review

(a) Many sports games have referees. What is the role of a referee in a game such as baseball or basketball (or some other sport)? How is the role of the referee similar to that of the moderator described in this section?

1.3 Running a Game

This book is supported by the Freely Available Network Game (FANG) Engine. FANG is a library, used by the Java programming language. FANG is a game moderator. You will write games by specifying components, providing those components with rules, and then having FANG run its game loop over and over again, updating the state of the game according to the rules provided.

This section, in an effort to whet your appetite, presents the code for a simple game along with the commands you would type at the command-line to run the game. The goal is to make a game run rather than for you to understand how the game works; the rest of the book covers the same ground with detailed explanations.

A *Java program* is a description of rules written in the Java programming language. The Java programming language is written in words, some of which are special *keywords*, or words defined and required by the language, and some of which are *identifiers* or names for things which the programmer can choose.

To read a Java program, know that code that comes after a *//* on a line or between */** and **/* is a *comment*. A comment is ignored by Java (it is only there for us humans). In this book, comments are typeset in *italics*:

```
// this is a single line comment - next line is a blank line

/* <- that symbol (both characters together) starts a comment.
The comment is treated as blank lines by Java (content ignored).
The comment ends with that symbol -> */
```

In this book, Java keywords are typeset in **boldface**. Identifiers and all of the punctuation are typeset in a normal face. The following listing shows two keywords, an identifier, and a *block*: a block is any section of the program enclosed inside of curly braces, { and }. When to use a block will be explained in the next chapter.

```
public class DemonstrationClass {
}

```

A complete Java listing using the Game type defined in FANG along with two *spacesprites* types, the Ship and the Meteor, is given below.

```

1 import spacesprites.*; // get all space sprites
2 import fang.core.Game; // FANG game moderator - has game loop
3
4 /** Asteroids-clone game */
5 public class Meteors
6     extends Game {
7     private Ship playersShip; // Ship = component type for game
8     private Meteor rock; // Meteor = component type for game
9
10    //Called (automatically) before the game starts by FANG
11    @Override
12    public void setup() {
13        playersShip = new Ship(this); // get new component from system
14        playersShip.setLocation(0.2, 0.8); // id.command(parameters)
15        // what to shoot, what to run into
16        playersShip.setTarget(Meteor.class);
17        playersShip.addCollision(Meteor.class);
18        // show the score and remaining life counter
19        playersShip.showLives();
20        playersShip.showScore();
21        addSprite(playersShip); // add ship to game
22
23        rock = new Meteor(this); // get new component
24        rock.setLocation(0.8, 0.8); // position
25        addSprite(rock); // add rock to game
26    }
27 }

```

Listing 1.1: Meteors. java

Important things to notice: the name after the keyword `class` in line 5 matches the name of the file where the code is stored (without the `. java` at the end); there are two nested blocks, the outer one running from the end of line 6 all the way to line 26 and the inner one running from the end of line 11 to line 25; and an identifier can be asked to perform an action (of which it is capable) by having the name of the identifier, a dot, the action, and any information needed by the action in parentheses.

To make a Java program run is a two step process: first you must *compile* (translate) the Java program into a simpler form and then you must *execute* (start) the simpler form with the `java` program. The following commands assume that you have a *command prompt*, a computer shell program where you can type commands. In the following, the *prompt*, the characters printed by the computer to tell you it is ready for a command, appear in italics, what you type is shown in normal type.

The prompt shows the folder where you are; the tilde, `~`, is the current user's home directory (folder) so the source code for the class was installed directly below the home directory of the user of the code. The `-classpath` parameter includes a dot, a colon, and then the *ppath* (folder location) of the `fang.jar` file where the FANG library code is provided.

```

~/Chapter01% javac -classpath ./usr/lib/jvm/fang.jar Meteors.java
~/Chapter01% java -classpath ./usr/lib/jvm/fang.jar Meteors

```

Assuming all goes well with typing in the `classpath` (where Java looks for library information used by the program), the first command will result in a slight pause (while the `javac` program, the Java *compiler*, translates the code we see in the listing above into a simpler language which is machine-friendly) and the prompt will be printed again. No news is good news.

The second line runs the `java` (not `javac`) program. When the program runs, then you should see a window similar to that in Figure ??.

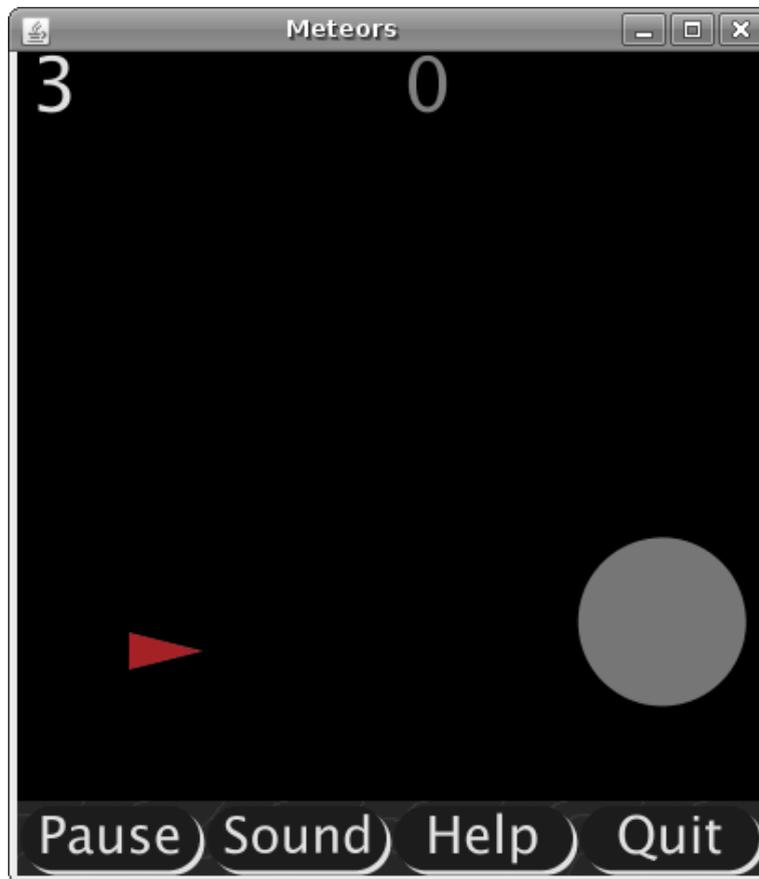


Figure 1.4: Meteors: A running FANG game

In FANG, everything above the four buttons at the bottom of the window is the game canvas; FANG game objects such as `ship` and `rock` are drawn on the game canvas. The triangle in the lower-left of the window is ship. Looking at the code in Listing 1.1, the location of `ship`, set in line 13, is $(0.2, 0.8)$.

FANG objects are located by their center point. The center of `ship` is one-fifth of the window width from the left edge of the window (0.2) and four-fifths of the window height from the top edge (0.8) . The width and height of the canvas are *always* treated by FANG as 1.0 and 1.0 , no matter what size the window actually is. $(0.0, 0.0)$ is the upper-left corner of the window and $(1.0, 1.0)$ is the lower-right corner (the *y*-axis is *inverted*; values get bigger the further down the screen you go).

A FANG game is started by clicking on the **Start** button just below the game canvas on the left. When `Meteors` begins, the meteor begins moving (its speed and direction are randomly selected). The ship is controlled by the keyboard with left and right arrows turning it, the up arrow applying thrust in the current direction, and space bar shooting little blue lazer balls that shatter meteors into smaller and smaller pieces.

The two numbers at the top of the screen are a count of the number of lives remaining for your ship (upper-left corner) and the score (one point per hit on a meteor). The game keeps going even after all meteors are destroyed or the ship is destroyed. Detecting the end of game and doing something reasonable in that circumstance takes more code.

The one thing we will try changed, though, is the number and size of the meteors. Listing 1.2 creates three meteors.

```
1 import spacesprites.*; // get all space sprites
```

```

2 import fang.core.Game; // FANG game moderator - has game loop
3
4 /** Asteroids-clone game */
5 public class MeteorShower
6     extends Game {
7     private Ship playersShip; // Ship = component type for game
8     private Meteor rockA; // Meteor = component type for game
9     private Meteor rockB; // Meteor = component type for game
10    private Meteor rockC; // Meteor = component type for game
11
12    //Called (automatically) before the game starts by FANG
13    @Override
14    public void setup() {
15        playersShip = new Ship(this); // get new component from system
16        playersShip.setLocation(0.2, 0.8); // id.command(parameters)
17        // what to shoot, what to run into
18        playersShip.setTarget(Meteor.class);
19        playersShip.addCollision(Meteor.class);
20        // show the score and remaining life counter
21        playersShip.showLives();
22        playersShip.showScore();
23        addSprite(playersShip); // add ship to game
24
25        rockA = new Meteor(this, 4); // get new component
26        rockA.setLocation(0.8, 0.2); // position
27        addSprite(rockA); // add rock to game
28
29        rockB = new Meteor(this, 2); // get new component
30        rockB.setColor(getColor("yellow"));
31        rockB.setLocation(0.2, 0.2); // position
32        addSprite(rockB); // add rock to game
33
34        rockC = new Meteor(this); // get new component
35        rockC.setColor(getColor("misty rose"));
36        rockC.setLocation(0.8, 0.8); // position
37        addSprite(rockC); // add rock to game
38    }
39 }

```

Listing 1.2: Meteor Shower. java

The three lines for constructing a meteor are about the same as they were in the previous listing. The only difference is that two meteors, `rockA` and `rockB` are constructed with different sizes: the rock in the last game (and `rockC` in this game) started at size 3; they broke in half three times. Sending in 2 or 4 (as in lines 25 or 29) changes the starting size and the number of levels of splitting.

Lines 30 and 35 show a new command which Meteor supports (all screen objects in FANG support it; you can try changing the color of `playersShip` if you like) is `setColor`. Our game has a named rule, `getColor` which, given a name of a color in quotes, returns the color. This then goes to the `setColor` method to change the color of the object on the screen. Figure 1.5 shows the result of a few moments play, all the meteors scooting around.

This section is just a hint at what you can do with FANG. One of the goals of the text is to get you started with all the eye-candy and support in FANG, to get you playing with Java. Then, as the book goes on, you will learn how to get past using FANG and understand Java much better. The important thing to do is to start playing with the code.

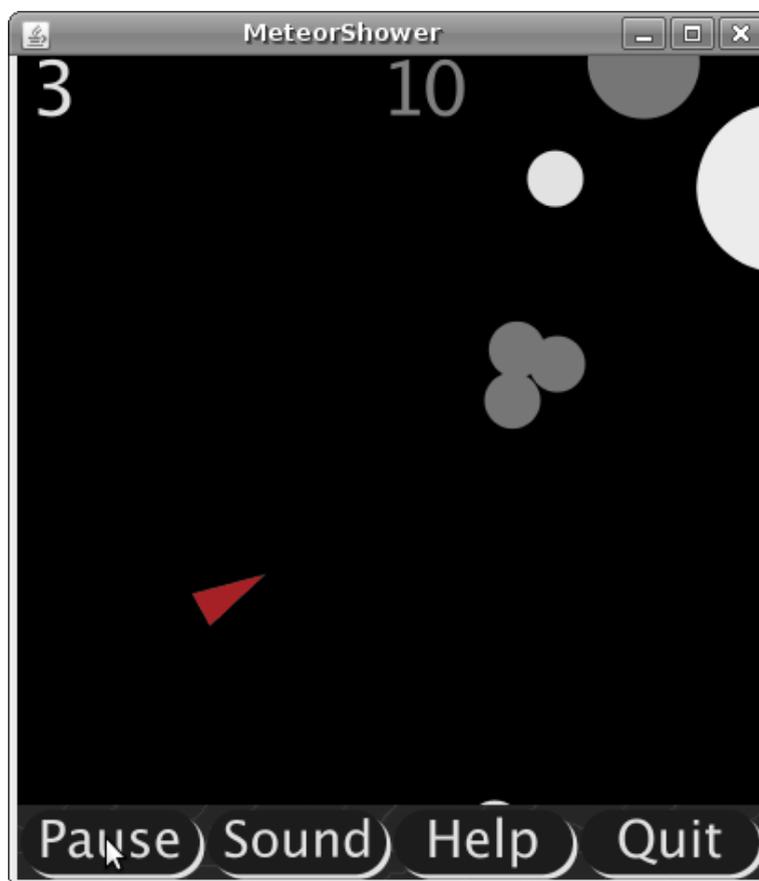


Figure 1.5: Meteor Shower: Game in progress

Review

- Looking at Listing 1.4, what line number would you change to have the `playerShip` start in the upper-left rather than the lower-left corner? What would be an appropriate change?
- What do you think you would change in `Meteors.java` to change the gameplay so that the meteor does not collide with the player's ship?
- Assume you wanted to make major modifications in `Meteors.java`, so major that the result would really be a different game. Assume you copy the Java file to `NewSpaceGame.java`. What is the very first change you would have to make so that the program could compile?

1.4 Strategies: Winning a Game

A strategy is a manner of playing a game. Just as a game has components and rules, a strategy has a game and a set of rules. The rules here are not the rules for *playing* the game but rather the rules for *winning*² the game.

²There are valid goals in playing a game other than winning: having a good time, socializing, killing time, etc. A strategy could be devised to support any of these goals; in this section we assume a strategy is selected to win the game.

Systems: *All of the Components*

If we take a step back from the game and consider the game plus all of its players, we have a system where each player is an active component (and a rule follower), following their own rules looking to win the game. The rules for a strategy are probably more complex than the rules for the game they play³ because they likely include the rules of the game. That is, a winning strategy in checkers must reflect a knowledge of what constitutes a legal move as well as a sense of what constitutes a *good* legal move.

Abstraction: Different Levels

In a game-mastered checkers game and its players, there are three active components and thus three rule followers: the moderator follows the rules of the game and the players each follow the rules of their individual strategy. As anyone who has mastered any non-trivial game can attest, developing a winning strategy is exceedingly difficult. While we define strategies here, it will not be until much later in the book that we try to computerize them, describing strategies to computer rule followers. This section just expanded our view of a game system to include players and consider them as active components following a set of their own rules.

Review

(a) Tic-tac-toe is a simple game with some basic strategies. Using your own personal experience or through some internet research, describe two strategies you can use to help you win at tic-tac-toe.

1.5 What is in a Computer Program?

A computer program is a collection of data or information stored in the computer and instructions that tell a computer "what to do". That is, it consists of components and rules which together interact with the user to permit them to accomplish something. This definition depends on understanding what a computer *can* do and how it can be instructed. We must take a moment and define what a computer is. This definition of a computer is going to be abstract, leaving more detailed discussions until they are necessary.

What is in a Computer?

At a minimum, a computer consists of a memory and a processor. In order to communicate with the outside world, the computer also needs some form of input/output (I/O) devices like keyboards, mice, joysticks, and a video monitor.

Rules and Components: Memory

The memory of the computer can be random access memory (RAM), a CD/DVD, or optical, drive, a hard drive, or a memory card. The main differences between the memory types are speed, cost, and size. Faster memory such as RAM is more expensive per unit amount and tends to be smaller while slower memory such as a hard drive is cheaper per unit and tends to be much, much larger. Other differences include volatility: RAM memory contents last only as long as the computer is on; hard disk and memory card contents are nonvolatile and last even after power is turned off.

A typical computer advertisement will mention the computer having between 1GB and 4GB of RAM and a hard disk drive of 80GB to some much higher number. What is a "GB"? A gigabyte is a GB. Computer scientists and computer advertisers disagree on what a GB is. Advertisers use powers of ten and the giga- prefix means one billion or 10^9 bytes. Computer scientists use powers of two and the giga- prefix means 1024 cubed ($1024 = 2^{10}$ so giga- means 2^{30}) or 1073741824 bytes. A single Western European alphabetic character takes up a single byte.

What do these various memories remember? The memory of a computer is used to store computer programs and the data that the computer programs operate on. That is, the memory contains the program that

³Exceptions such as "Pick any legal move at random." are unlikely to produce very *good* winning strategies.

runs Quake as well as the description of your character, the other characters and objects in the level, and the current level. That is, the memory contains rules and components.

Rules Follower: The Central Processing Unit

With rules and components represented in the computer's memory, the only thing missing from our earlier game system description is some rule follower to handle any active components represented in the computer. The processor or central processing unit (CPU) is a rule follower.

The CPU is advertised with the number of *cores* or complete processors that are on the chip at the center of the computer as well as the clock speed of each core. The clock speed or clock of the core is approximately the number of instructions the core can execute per second.

Different kinds of CPU chips use different instructions (thus the machine instructions for an Xbox 360™ are different than those on an Intel-based PC, even if the two might run the “same” game). The exact structure of the instructions is beyond the scope of this book though we will come back to computer instructions in the next chapter when we talk about what a computer can possibly do for us.

Modern CPUs run at a given speed, the gigahertz of the computer. During each time step or *clock cycle*, the CPU runs one iteration of its own fetch-execute-store loop.

The Central Loop

The instructions are stored in the RAM. Each clock cycle the CPU fetches an instruction from the memory, determines what the instruction is, executes the instruction (which may require fetching data from the RAM), and stores the result back to the RAM. This fetch-execute-store loop is the central loop in the computer. No matter what program you are running, this is what is happening hundreds of millions or billions of times per second.

The above is a simplified view: fetching information from secondary storage takes much more than one clock; fetching information from RAM often takes more than one clock; some CPUs can execute more than one instruction per clock cycle. We will stick with the simplified view because it is sufficiently close to the real thing to give us insight into how the computer works.

After the current instruction, what instruction does the CPU fetch the next time through this loop? This is one of the incredible powers of modern computers: it typically continues with the next instruction in order in the RAM but it *can* change the next instruction to any other instruction in memory depending on results of some calculation. This ability to select what to do next, combined with the incredible speed of execution, makes the modern computer a general purpose information processor (and game platform).

Talking to the World: Input and Output

The minimal computer described above is also lacking interaction with the real world. That is, input from and output to the player of the game. Output is presented on a computer screen and through computer speakers. These output devices use the memory of the computer and often their own processor to present pretty pictures and music to the user.

The keyboard, mouse, and even a joystick can provide input to the computer, input that the CPU can react to. The input devices provide events that the CPU can react to.

Software: Rules and Components

A computer program is a collection of components stored in a computer's memory. Those components interact according to rules also stored in the computer's memory. The rules are followed by the computer's CPU and each rule can result in changes in the state of components stored in memory.

Multitasking

Interestingly, a computer program can have multiple rules being followed at the same time even if the computer has only a single CPU. How? Consider our game-mastered checkers game again. Imagine that rather

than two players playing checkers there are eight players who wish to play checkers. How could one moderator satisfy all of the players?

The moderator could line up the players and play a game for the first pair and then, when that game is done, play a game for the second pair, and then help the third and fourth pairs. By serving each pair *sequentially*, the moderator has helped four pairs play checkers. Unfortunately, though, it takes four times as long as playing one game to finish all four (assuming, for the moment, that all checkers games last the same amount of time).

An alternative scheme would be for the moderator to call three other moderators so that all four games could be played, in *parallel*, using four identical sets of components. This approach requires only the amount of time it takes to play one game to finish all four games; it is quite costly in terms of checkers, boards, and moderators.

Consider a hybrid approach: have four sets of checkers components (pieces, boards) but only one moderator. The four sets of checkers represent the four games and the four pairs can play simultaneously but the moderator divides his time between the four games. That is he can ask the player in game one for his move and then, while waiting for the answer he can make a pending move in game two, announce a victory in game three, check that the player in game four has made a legal choice of checker and then return to listening for a move in game one.

This division of attention takes advantage of the fact that in the original game the moderator spent most of his time waiting for things to happen. Using what in one game was waiting time to service a different game means that four simultaneous games take only a little bit longer than a single game. This is an example of *multitasking*, working on multiple tasks by using down time in one task to take care of another task.

Multitasking is not getting something for nothing. Imagine very fast players selecting moves as quickly as the moderator can make them. This would take all the moderator's time just to service *one* game. If all four pairs were that fast, multitasking would actually take *longer* than running the four games in sequence. Multitasking might still be a good idea as the last pair in line would get to start their game sooner than if they had to wait for the three other games to finish.

Similarly, a single CPU can multitask, providing a computer program with multiple parallel rule followers, each getting a slice of the attention of the CPU. This is important because a single computer program might need multiple active components. Imagine a program that reads from the keyboard and receives information from the network at the same time; each of these activities should have its own rule follower so that no input is missed.

A rule follower in a computer program is also known as a thread of control. Will a program with multiple threads of control run many times slower than one with a single thread of control? Yes and no. If all of the threads of control have very complex sequences of rules that work on very small components, then it is possible that each thread of control is trying to use the full capacity of the CPU. Such threads are *processor bound* and this situation is equivalent to the players making moves as fast or faster than the moderator.

Most programs spend at least some of their time waiting for events that occur slowly, in terms of CPU speeds: waiting for data read from the network, from a hard drive or optical drive, waiting for a human being to press a key. Human reaction speed is hundreds millions of times slower than the time it takes the CPU to follow a single rule; a lot of work can be provided to a ready thread while another thread waits for an event.

While multiple threads of control will not execute as fast as if the machine were truly parallel (a CPU per thread of control), it will run much faster than if each thread of control were executed from beginning to end one after the other. It also has the advantage that the different threads of control can communicate and cooperate to provide the user with a compelling experience.

Levels of Abstraction

A computer game is both a game and a computer program. A computer game is two *different* collections of components and rules. It has components and rules at two different *levels*.

Designing a computer game is really two different design tasks: designing a game and then translating the components and rules of the game into virtual components and rules in a particular computer programming language.

Translation typically involves moving from a very high level of abstraction to a more concrete level by specifying more details. This section introduces some of the general techniques for specifying rules and com-

ponents in computer games (and other computer programs, too). The remainder of the book can be read as the continued expansion of techniques mentioned here.

Designing a game requires defining the components and the rules. It is possible to enumerate the components⁴. Rules, however, are more difficult to define. A first thought might be to list all of the configurations of the components and how to get from one to another. Unfortunately, the number of configurations for even a small number of components can grow astronomically: with a 3x3 grid, each empty or containing an “X” or an “O” there are almost twenty thousand component configurations⁵. Some of the configurations are illegal in play (the board containing nine “X”s is counted though you could never see it in play) but the exponential growth in configurations means that rules must be expressed more compactly.

Instead of exhaustively listing all game configurations, simple rules are typically built up into more complex rules. The rules for combining rules can be considered *problem solving techniques*⁶. Very simple rules (*i.e.*, pick a checker, capture a piece, roll two dice) are combined into more useful rules using five simple techniques: sequence, selection, iteration, delegation, and abstraction.

For example, in Listing 1.2 on page 7 the rule follower asks a player to select a checker and then select a spot to move the checker and then the checker is moved. This simplification is an example of the simplest rule writing technique: sequence. Do this *and then* do this other thing *and then* do that. This is the simplest mechanism for combining simple rules into more complex rules.

The previous example is an oversimplification of the original processing done by the moderator. A player selects a checker and the moderator determines if it is legal or not. If it *is*, then the moderator asks for a target square; if it is not, the game master asks again for a checker to move. This illustrates the second rule combining mechanism: selection. Two rules can be combined with some condition that can be tested and one or the other of the rules will be executed but not both.

The moderator’s complete program actually repeats a sequence of two selection statements and a smaller sequence multiple times. Such repetition combines rules using the iteration rule writing technique: a rule can be combined with some sort of condition and the rule will be repeated until the condition is met (or so long as the condition is met; these two are logically indistinguishable).

The additional rule writing techniques were not used in the game master’s “program”. Delegation is the creation of a new named rule. It would be possible to use the following definition:

```
define move for player
  ask player to pick a checker
  if checker is not legal fail
  ask player to pick move
  if move is not legal fail
  make move
  succeed
```

This defines a new rule for the moderator, a rule called `move`. That rule can be applied to any player and will be applied to each player in turn:

```
while game not done
  if move for current player succeeds
    change current player
```

This new rule shortens the moderator’s main routine at the cost of defining the `move` command. When you learn to name new commands in Java there will be more discussion of when it makes sense to use delegation and when it makes sense to include code directly *inline*.

The final rule combining technique, abstraction, is the packaging of an entire game, a “mini-game” as a component in another game. Examples of this will be provided as we study how to define our own types in Java.

The earlier description of checker’s components brings up one possible problem in describing a game’s components: scale. That is, is the checkerboard a component that is divided into squares (the squares are

⁴*infinite* games are beyond the scope of this book

⁵ $3^9 = 19683$

⁶These problem solving techniques are based on Nell Dale’s problem solving techniques from the Turbo Pascal days (see [DW92])

part of a component) or are the squares components that are assembled into a checkerboard (the board is a collection of components rather than a component itself). The answer to the question is one of scale and picking the right scale to examine a game (or a computer program) is something of an art rather than a science; both of the above views are valid and it might be nice to be able to switch between them at various moments (when moving a given checker it seems the squares are important; when selecting a checker to move the checkerboard's configuration is most important).

Interpreting Memory

The RAM contains both instructions (fetched by the CPU in the fetch-execute-store loop) and data (loaded in the execute phase and then replaced in the store phase). How can you tell what a given location in the RAM *means*? The answer is that you can't. That is, a given value in a given place in the RAM only has a particular meaning if you know the context in which it should be interpreted.

The need to know how to interpret a memory location has implications in how we write computer programs: when we set aside memory to hold data we must tell Java what *type* of component we will put in that location. That way when we read or write the actual values stored there we know the context in which it should be interpreted. We will return to this when we begin assigning types to memory locations.

A computer program is a collection of virtual components that reside in the computer's memory and interact according to a collection of rules also stored in the computer's memory. The rules are followed (active components are executed by) the computer's processor. A single processor can be shared between many active components (threads of control) because most active components have pauses built into their rules sequence (waiting for input from the user or reading from one of the slower components in the memory) and those pauses can be used to overlap execution of multiple active components. Active component's sequencing can be modified by events that the computer detects. For game purposes the primary events of interest come from input devices such as keyboards and mice or from other computers connected to the network.

Review

(a) Between RAM and hard drives, which tends to have a larger capacity? Which tends to be more expensive per gigabyte (GB)?

(b) Consider eating at a small restaurant with only 4 tables and 16 chairs. Is it necessary to hire 16 waiters to serve each patron individually? Why not? About how many waiters would be required? How does a restaurant with fewer than 16 waiters serve all 16 customers who may be in the restaurant at the same time? (insert footnote) There is a video game called Diner Dash where aspiring waiters can practice.

(c) Do some internet research to find out how much memory a computer used for gaming needs. Cite at least two sources in coming up with your answer.

(d) Describe how to draw the letter A using a combination of only the following instructions (where X can be replaced by a number):

```
put pen down
lift pen up
move pen forward X cm
turn pen X degrees
```

Assume the pen starts up and after drawing the letter, the pen should finish up.

(e) Take your answer to the previous question and randomly reorder the instructions, then follow them. Do you get an A? Why not? A similar thing might happen if you happen to place the correct set of instruction for a computer program in the wrong order.

1.6 Summary

In this chapter we developed a simple definition for a game: a *game* is a collection of components that interact according to a set of rules. Components are the *things* used to play a game. Rules describe how the components of the game *interact*.

Rules specify the starting configuration of components, legal moves to change the configuration, how the configuration is evaluated for scoring, how to know when the game ends, and how to determine the winner.

The same game can be played with different components: whether you use poker chips, different denominations of coins, painted manhole covers or bottle caps, the movement and capture rules make the game checkers. Alternatively, multiple games can be played with the same components interacting according to different rules: solitaire, poker, go fish, or rummy are all played with the same deck of cards but they are very different games because the specified rules are different.

A *rule follower* (moderator) can automate the rules of a game. The rules for playing the game are distinct from a *strategy* for winning the game. A rule follower only knows the rules of the game and how to move it forward. Each active component can be considered to have a rule follower assigned to it (even rule followers which just follow the laws of physics).

A *computer program* is a collection of components that interact according to a set of rules. Both components and rules are stored in the computer's memory (RAM). Rules are followed by the computer's processor (CPU). Rule followers in a computer program are called *threads of control*. A single processor can simulate multiple threads of control by rapidly switching from one thread of control to another, giving each a tiny slice of its attention. The result is *multitasking* which can make use of otherwise wasted CPU time.

Rules, for a game or a computer program, are almost always built up. Simple, single action, rules are combined using the five problem solving techniques: sequence, selection, iteration, delegation, and abstraction. Sequence is doing one thing after another. Selection is evaluating some condition and doing one thing or another depending on the condition's value. Iteration is repeatedly doing something until some condition is met. Delegation is naming a part of a program to make a new command. Abstraction is the creation of a new component, a component that encapsulates its own rules into a sort of mini-game within a game.

Chapter Review Exercises

Review Exercise 1.1 Describe the components and rules of Tic-Tac-Toe.

Review Exercise 1.2 Describe the components and rules of Chinese Checkers.

Review Exercise 1.3 Describe the components and rules of Go Fish!

Review Exercise 1.4 Describe the components and rules of poker.

Review Exercise 1.5 Compare the components and rules from the previous two questions.

Review Exercise 1.6 Ignoring whether a position is legal or not and assuming an unlimited number of both colors of checkers, how many board positions are possible on a checker board?

Review Exercise 1.7 Consider the game Tic-Tac-Toe. Describe what a rule follower (moderator) would do to supervise a game between two players.

Review Exercise 1.8 Consider the game *Battleship*⁷. Each player begins with a 10 × 10 grid of squares and a fleet of five ships of varying lengths (lengths appear after each ship's name): aircraft carrier(5), battleship(4), cruiser(3), submarine(3), and destroyer(2). To set up the game, each player places their fleet in their grid, each ship placed horizontally or vertically across the given number of squares.

⁷A version of *Battleship* is produced by Hasbro Games but the game predates the Milton Bradley Games (now Hasbro) 1967 release by at least 50 years.

Players then take turns selecting squares in their opponents grid, shooting at any ship occupying the square. The goal is to sink the enemy's fleet before a player's own fleet is sunk.

Formally describe the components and rules for this game.

Review Exercise 1.9 How would you add a rule follower to *Battleship*?

Review Exercise 1.10 If the ships and the grid are the components of *Battleship*, consider modifying the rules:

Describe how you would modify the rules to make the game more difficult.

Describe how you would modify the rules to make the game easier.

How would you modify the rules to permit players of different skill level to compete evenly (be sure to note your definition of even).

How could you use the same components to play a completely different game?

Review Exercise 1.11 For *Battleship*, it is possible to vary the components while keeping approximately the same rules.

Describe how you would modify the components to make the game harder.

Describe how you would modify the components to make the game easier.

Can you make the game work between opponents of differing skill levels by modifying just the components?

Review Exercise 1.12 You have been asked to stage a game of checkers during the half-time of a football game in a large, outdoor stadium. Describe the components you would use. Would there be any need to modify the rules of the game?

Designing Your First Program

A game is a collection of components with associated rules governing their interaction providing the user with meaningful choices to solve the problem presented by the game environment and/or an opposing player. Game components and rules can be modeled and executed inside the memory and processor of a computer.

This chapter introduces Java, a modern, high-level programming language which can run on multiple platforms. To support simple computer games in Java, it also introduces the Freely Available Networked Game (FANG) Engine, a pure Java framework. Your first computer programs will focus on the structure of all FANG/-Java programs and how to create visual components using FANG. The next chapter moves beyond just drawing pictures to defining rules for a complete game.

2.1 Drawing a Playing Card

Each chapter from here on begins by designing a simple computer game. As the design is fleshed out, it will become clear what we need but do not yet know how to do. The remainder of the chapter will then present the Java or FANG features necessary to implement the design and the chapter will wrap up with a return to the original design showing how the new features apply.

Our first “game” will be drawing a four of clubs on the screen. Just as movie directors often begin with a storyboard for their films, we will typically begin with a sketch of how the game will look on the screen. We will include notes on the screen (or next to it) which discuss how different parts of the screen move or interact.

Figure 2.1 illustrates how a four of clubs would be drawn on the screen, one club near each of the corners of the playing field. The club in the upper-left corner is drawn with just outlines of its constituent shapes. Along with the note, it shows that a club, while somewhat complicated in appearance, can be decomposed into four very simple shapes: three circles for the leaves and a rectangle for the stem.

A Pre-game Program

Look back at our game definition. Since a game must present the player with meaningful choices the game must respond to different player choices in different ways. This requires, at a minimum, being able to select one or another set of rules depending on what the player does. This chapter will focus on setting up game components on the screen, leaving the main video game loop to the next chapter.

In Java, sequence, the simplest way of combining primitive rules, is indicated by the order in which statements are written in the source code. That is, in order to do A and then do B, we simply write A and then, on the next line, write B.

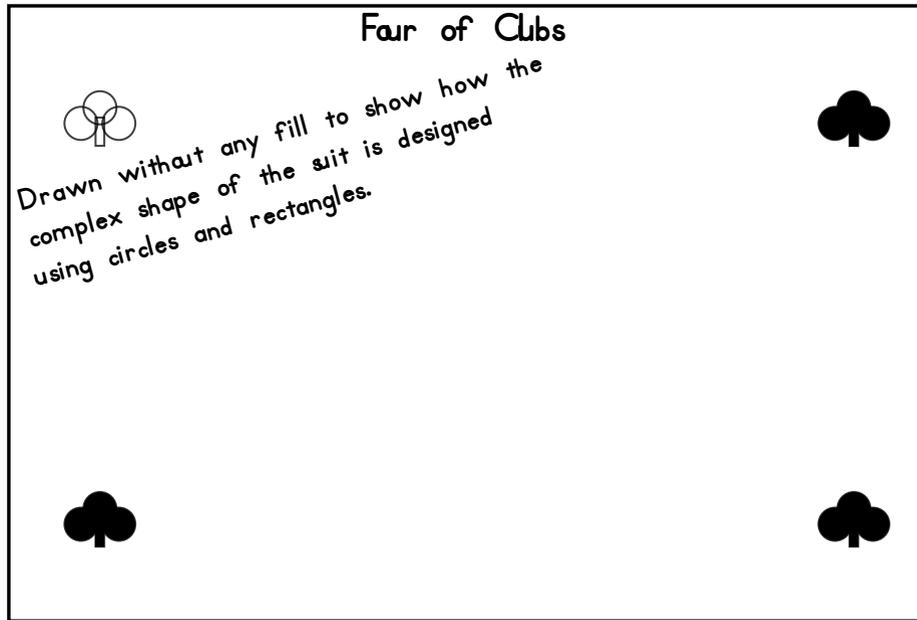


Figure 2.1: FourOfClubs game design diagram

Creating Components

In Java, components have types. In order to use objects of most types you need to create an *instance* of that object. An instance of an object is created by using the keyword **new** and the name of the type of component you want to create.

This is analogous to the idea of a *Pawn* type for a game of chess. The rules of moving any pawn are part of the definition of the type while the color, starting location, and current location of any *instance* of a *Pawn* is specific to that instance. You can imagine that a pawn is created by specifying the color and starting location: *new Pawn("white", "f2")*. There would be sixteen such creations, eight for the white pawns on row 2 and eight for the black pawns on row 7.

In Java parlance, the call of the type name after **new** is a call to a *constructor* for the class. Just as with the *Pawn* example above, a Java constructor can take *parameters* which determine specific things about the instance being constructed. Thus, it is possible to construct a rectangular component in the middle of the game screen with the following constructor call:

```
new RectangleSprite(0.5, 0.5)
```

The two parameters in this constructor represent the width and the height of the rectangle, measured in screens; there is more on what a *sprite* is and screen coordinates in Section 2.5. Before going into details on all the parts that make up a FANG game program, we will look at a very short example, one that is available along with the book.

A Whole Program

Listing 2.1 is a very short example program. In short, the program defines a special type of *Game*, one called *Square*. The rules of *Square* are a single *setup* method which creates a rectangle using **new** and then adds the rectangle to the game using *addSprite*.

```
1 import fang.core.Game;
2 import fang.sprites.RectangleSprite;
3
```

```
4 // FANG Demonstration program: RectangleSprite
5 public class Square
6     extends Game {
7     // Called before game loop: create named rectangle and add
8     @Override
9     public void setup() {
10        RectangleSprite rectangle = new RectangleSprite(0.5, 0.5);
11        rectangle.setLocation(0.5, 0.5);
12        addSprite(rectangle);
13    }
14 }
```

Listing 2.1: Square.java: A complete program

The lines beginning with `//` are comments, intended for human readers of your code. The line with `@Override` is a special direction to Java that the `setup` method here specializes the general method defined for all games.

The results of compiling and running `Square` is shown below in Figure 2.2. The screenshot is in grey scale; the actual square is in a color referred to as FANG Blue.

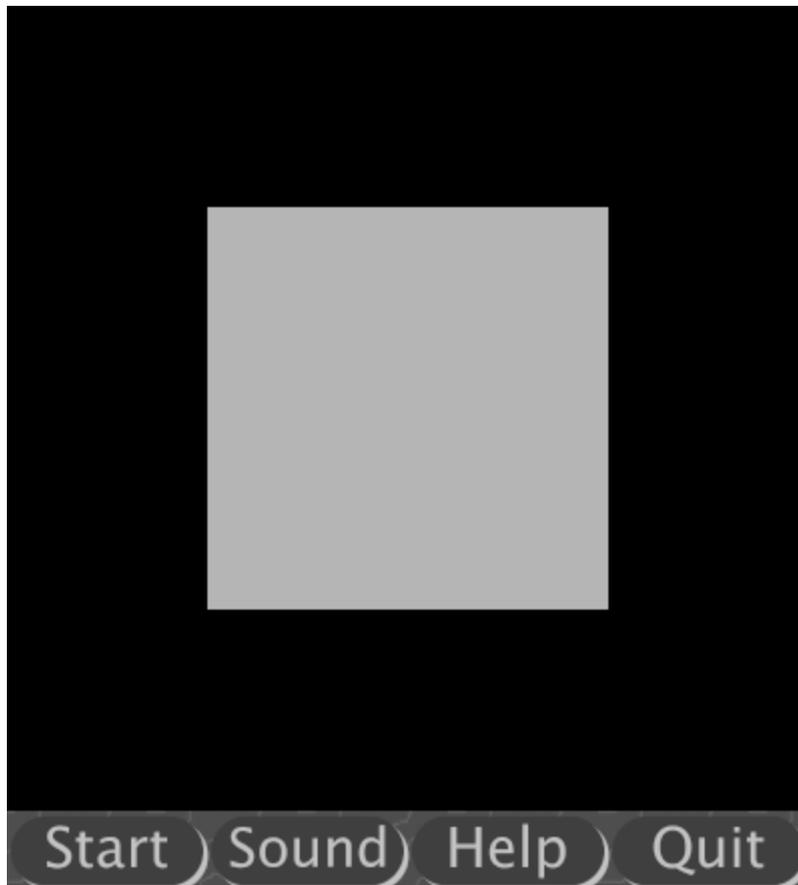


Figure 2.2: Running Square

The next section will delve deeper into what a computer is and how it works, leading to the following section's discussion of the Java programming language. At that point the general layout of Java/FANG programs

will be presented along with how to make different shapes and different colors in a FANG program. Having drawn several different pictures using FANG programs, we will return to the four of clubs designed at the beginning of the chapter.

Review

- (a) Write the Java/FANG code to construct a `RectangleSprite` as wide as the screen and 75% as high as the screen.
- (b) What is the purpose of the `//` characters in a Java program?

2.2 How Computers Work

This section detours away from designing a game and translating our design into a program. It looks at what a computer is in more detail than we saw in the last chapter. It then goes on to discuss how a computer program executes on the computer hardware and the exact mechanism of programming a computer.

Digital, General-Purpose Computers

The computer we program is a digital, binary, general-purpose computer. Digital means that the computer stores and operates on discrete rather than continuous values. Consider the difference between an old-fashioned, *analog*, needle speedometer (as in the left of Figure 2.3) and a speedometer with a *digital* read out (as in the right of the figure). The needle can represent any speed between 0 and the maximum reading; if you speed up just a little bit, the needle will rotate just a little bit clockwise, updating the reading of your current speed.

To emphasise the difference between digital and analog display, assume the digital speedometer has the same range as the analog meter but that the ones digit is always “0”. That is, the digital speedometer shows speed to the nearest ten miles per hour. How many different values can the digital speedometer display? Assuming either speedometer is limited to the range 0-100, the digital speedometer can show only eleven different speeds. This limited number of discrete values is compared to the continuous resolution of the analog device.

Digital electronic signals (like those inside the computer) are more robust than analog signals. It is much simpler to build a circuit that can differentiate between the presence or absence of 3.3VDC than it is to build a circuit that can differentiate between all the values between 0 and 3.3VDC. This also explains why modern digital computers are *binary* (using the base-2 number system; more on this later): it is easier to differentiate between 0 and 3.3VDC rather than being able to tell 0 from 0.33 from 0.66 from 0.99 and so on to have eleven discrete values.

While there have been analog computers, most modern computers are digital. Each memory location in the computer can hold a number of discrete states. You might ask how a machine, limited to holding entries from a small set of possible values, can be a general-purpose machine. The answer to that question comes in two parts: how many states can a sequence of discrete entries take on and what do those states mean?

Note that you are holding a book in your hands, one written in the English language. Ignoring the pictures for the moment, how many different symbols can any given location in the book contain? That is, counting from the beginning of the book, what value might the one thousandth character in the text have? The ten thousandth? Any, randomly selected location? There are only about 75 different values possible (a space, 26 lowercase letters, 26 uppercase letters, 10 digits, and a handful of punctuation characters; this is a computer science book so the punctuation is more varied than in a book by Mary Shelley).

Yet, this sequence of alphabet characters, this *book*, purports to teach you about building simple computer games; no matter how simple the games are, the concept being taught is moderately complex. The sequence of characters, drawn from a fairly limited alphabet, conveys a great amount of information. And a different sequence drawn from the same limited alphabet might teach you how to bake fabulous deserts or tell you the story of *Frankenstein: or the Modern Prometheus*.

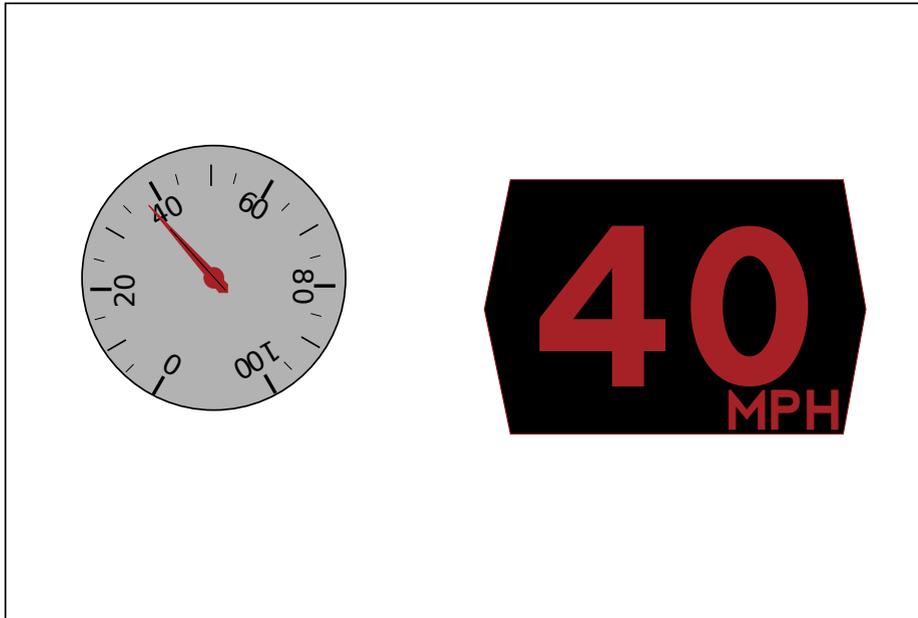


Figure 2.3: Analog v. Digital Speedometers

The same thing happens inside the computer: its memory, composed of *bytes*¹ which each draw their value from one of 256 possible values, is arranged as a large sequence of bytes. Modern machines have main memories (RAM) measured in *gigabytes*² or billions of bytes. To put that in perspective, a page of this book contains about five thousand characters drawn from an alphabet about a quarter of the size. That means a page could be stored in just over a thousand bytes. That means a computer with a one gigabyte memory could store almost a million pages in the RAM at one time.

Note that the memory inside the computer, the random access memory is not fixed like the ink on this page. Instead, it can change over time. Thus it can store a different selection of a million pages at any given moment and, by following instructions, it can change any of those characters on any of those pages, changing what million pages are stored in a split second.

While each byte can be considered a number on the range 0-255, the interpretation or *meaning* of the contents of memory depends on what part of the computer is reading the value as well as on the current instructions being executed by the CPU. We will now take a closer look at the parts of the computer.

Parts of a Computer

Figure 2.4 shows an abstract block diagram of a computer. The middle of the diagram is the Central Processing Unit (CPU). The CPU is, literally, the rules follower for the computer. It follows a very simple cycle: fetch the contents of a location in the main memory, determine what the contents of that location mean in terms of the instructions the CPU knows how to follow, fetch any contents of memory that are needed by the instruction, execute the instruction, and store the results.

The main memory is *random access* in the sense that any byte can be accessed in the same amount of time. Individual bytes are addressed with their distance from the beginning of memory (memory location 0); no matter how large a byte's address, it can be accessed as quickly as the byte at location 0.

What would a non-random access memory look like? Consider a similar addressing scheme, where each chunk of data was labeled with its distance from the beginning of the memory but have it stored on a reel of

¹A byte is eight binary digits (or *bits*) wide; each bit can contain a 0 or a 1 so the total number of combinations that a byte can hold is $2^8 = 256$. A byte can directly encode the numbers from 0 to 255 which can be interpreted in different ways.

²In computer science we use standard power of ten prefixes (i.e., giga-, mega-) to refer to the nearest power of two: a *kilobyte* is $2^{10} = 1024$ bytes; a *megabyte* is $2^{20} = 1048576$; a *gigabyte* is $2^{30} = 1073741824$.

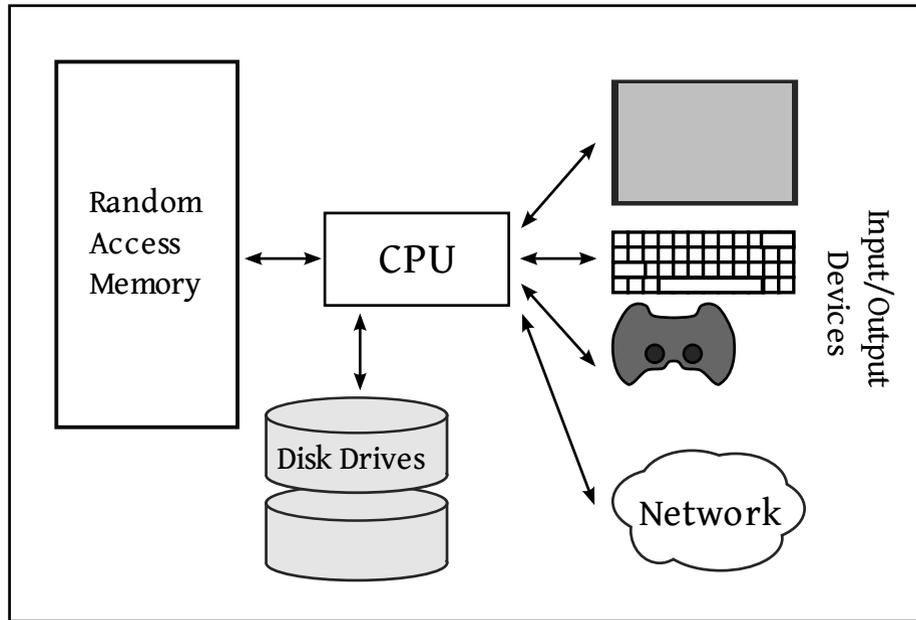


Figure 2.4: Abstract Structure of a Computer

magnetic tape. In the computer, the cost of accessing the byte at location 0 followed by the byte at location 1000000 followed by the byte at location 100 is three memory access times, all fairly quick.

Assuming the tape begins completely rewound, the same access sequence on the tape reel is very quick for byte 0 (the tape is positioned to read that location); then the tape must move 999999 memory locations past the read head and read the byte at location 1000000; finally, the tape must stop and rewind 9999001 locations to read the final byte. Even if moving the tape and not reading it is somewhat faster than reading each byte, moving forward and backward about two million locations certainly takes longer than it would to read locations 0, 100, and 200, one after another. In RAM, all sequences are equally fast.

The CPU can change the values stored in the RAM very quickly but the contents of the RAM are *volatile*: they go away when the power is turned off. Computers use *disk drives* to provide longer-term storage; disk storage includes optical (CD/DVD), magnetic (regular hard and floppy disks), and solid-state (memory stick, thumb drive) drives. These devices are slower but larger than the RAM. They are also *non-volatile* meaning that changes to the content persist across power-off cycles.

On the right side of the drawing are several *input/output devices* (I/O). These include the screen, the keyboard, game controllers, and even the network interface. These devices can provide information for a running computer program or be driven to display information from the running program for the computer user.

The instructions executed by the CPU are **very** simple. They are things such as: copy memory contents to a named location, store value in a named location, and if some condition, continue with a different next instruction.

For example, the following sequence of instructions would load the contents of two memory locations and then put the larger of the two values into a different location in memory³. The number to the left of each line is the memory location where that instruction begins (each is assumed to take exactly 4 bytes).

```

1248  load contents of memory location 100 to x
1252  load contents of memory location 104 to y
1256  if x > y jump to memory location 1268
1260  store contents of y to memory location 108
  
```

³The numbers in the example are base 10 rather than binary; in the computer the instructions and memory locations would be encoded in binary numbers stored in the bytes of memory

```

1264  jump to memory location 1272
1268  store contents of x to memory location 108
1272  ... <more instructions here>...

```

This sequence of six instructions makes sure that the value stored at 108 is the larger of the values at 100 or 104. It takes six instructions to do something a human could do on inspection. A modern computer is powerful because it can execute these instructions billions of times per second. Simple arithmetic and decisions, done quickly enough, permit the computer to run any program from an operating system to a word processor to a real-time strategy game. It all depends on the contents of the computer's memory.

Interpretation of Computer Memory

The contents of the memory do not, of themselves, mean anything. They must be interpreted and the same contents can be interpreted differently depending on context. That is no different from the contents of a book: if this book said it were in Italian or German, the sequences of characters would be interpreted differently (and, in this case, incorrectly); when the book is read by an editor or your instructor, their interpretation of the material is different than yours, reading it as a student of computer science.

What makes the computer truly general-purpose is that the contents of the memory can be interpreted as *instructions* to the computer on how it should manipulate or interpret other parts of memory as *data*. The fact that the contents of memory can be interpreted in different ways and that the contents of memory can describe *how* the contents are to be interpreted is what permits a general-purpose computer to store a song (encoded using the MP3 or Ogg Vorbis coding scheme, for example), a program to *play* a song (a machine-language program for manipulating encoded songs and the computer's sound hardware), and an operating system to *start* the program that can play a song (a different program for manipulating *programs* as well as the computer's memory and other hardware).

Viewing a computer at multiple layers simultaneously, where what is a series of instructions at one level (the player program) is treated as data at another layer (the operating system) is what makes modern computers so powerfully general-purpose.

John von Neumann [1903-1957]

John von Neumann was a prodigious mathematician who contributed to the development of game theory and computer science. He was born in Budapest, Hungary just after the turn of the Twentieth Century, received his PhD in mathematics by age 22, and emigrated to the United States in 1930.

In the US, von Neumann was an original member of the Institute for Advanced Study at Princeton University, one of the first places in the world to build digital electromechanical and electronic computers. He also contributed to the computing power, mathematics, and physics of the Manhattan Project, the United States' World War II super secret initiative to develop the world's first atomic bomb.

While consulting on the Electronic Discrete Variable Automatic Computer (EDVAC), von Neumann wrote an incomplete but widely circulated report describing a computer architecture which used a unified memory to store both data and instructions. This contrasted with the so called *Harvard* architecture where the program and the data on which the program operated were segregated.

The von Neumann architecture is the fundamental architecture used in modern computers. A single, unified memory makes it possible for a computer program to treat itself (or another computer program) as data for input or output. This is just what the operating system described in this section or the compiler describe in the next section does.

Von Neumann's report drew on but failed to acknowledge the work of J. Presper Eckert and John W. Mauchly; this led to some acrimony when it came time to allocate credit for the modern computer revolution.

Von Neumann's name will come up again when we examine game theory, a branch of economics/-mathematics that permits reasoning about strategies in competitive games.



Figure 2.5: Gandalf IMSAI 8800 at the Computer History Museum (1976 microcomputer) [IMS76]

How can a human being create a program that a computer can execute? One way would be to store the byte values for each instruction directly into the memory of the computer and then command the CPU to begin interpreting them as instructions. Figure 2.5 shows an IMSAI 8800 microcomputer; notice the two sets of eight switches. One is for specifying the contents a byte should have and the other is for specifying the address of a byte in the computer's memory. The user could key in a value, press the write key (one of the six control switches to the right end of the machine), increment the address switches, set the value for the next byte and so on.

Assuming we could get the CPU to execute them, this approach to writing a computer program is daunting: a modern game program may take up a megabyte (about a million bytes) for the executable; the rest of the DVD or downloaded package is typically art and sounds in various formats. Keying in a million bytes worth of instructions would take a long time (at one byte per second it would take just over eleven and a half days). The chances of building a correct program at that low-level are just about zero.

Better would be to write the program in a more compact, more abstract language; we refer to such a language as being at a *higher-level* than machine code. That language could be translated into machine codes which could then be entered into the computer. This is similar to outlining a paper before writing it: the outline permits you to solve problems with overall structure and the presentation of your argument without the distraction of overwhelming detail. Once your outline is correct, you can translate it into paragraphs, sentences, and individual words.

The high-level programming language used in this book is *Java*. Java, designed by Sun Microsystems in the 1990s, is an artificial programming language designed specifically to eliminate ambiguity. No ambiguity means that the translation from Java to machine language is by rote: a given sequence of instructions in Java will always translate into the same sequence of machine instructions. A major step in computer history was the realization that a computer program could do this translation (this is one area where Admiral Grace Murry Hopper made contributions). A *compiler* is a program which takes a high-level program as input and produces a low-level program which does the same thing.

Different CPUs are found at the center of different computing devices (game consoles, PCs, PDAs, cell phones, etc.) and different CPUs have different machine languages. As was said before, the contents of memory must be understood in context; the same sequence of values might mean something completely different in a different machine language. A higher-level program can be compiled for each CPU's machine language and then it can execute on different machines. Note that there must be a different machine language file for each different CPU.

Once a high-level program is translated into a computer's machine language, that machine code can be loaded into the computer's memory and executed by the *operating system*. Operating systems (such as Microsoft's Windows Vista, the open source Linux, and Apple's OSX) are an important field of study in computer science. For our purposes, they are programs that run on the computer that can start, stop, and manage *other* computer programs. Operating systems also permit programs to create directories (or folders), read and write files, and use the network. The operating system provides an environment in which our programs run and can access the various resources of the computer and the Internet.

By analogy, compiling is similar to translating an entire book from one language to another. If we consider translating an encyclopedia, translating the entire thing is a lot of work. The user of the translated encyclopedia may not even be interested in the entire text, just the article they look up along with those articles it leads them to. It might be more efficient to translate portions of the book only as they are needed. This would require an interpreter, fluent in both the source and target languages, alongside the reader but the translation time would be minimized.

This same approach can be used in the computer: a program in a higher-level language can be *interpreted*, each part being translated just in time for it to be used. Parts that are never used are never translated. Note that the interpreter, here a program like the compiler, treats the higher-level program as input along with the actual input processed by the program.

Compiling has a speed advantage over interpretation **if** the program will be run multiple times: the high-level representation of the program need only be translated once. Interpretation has a flexibility advantage if the program is going to change often: each time the interpreter is run, the most current version of the high-level representation is executed.

There is a hybrid approach where high-level programs are **compiled** into *bytecode*, a lower-level language that is not any computer's machine language and the bytecode representation is then **interpreted** by a program that understands bytecode and runs in the local machine language. Each bytecode is interpreted each time it is executed and it is still possible to have bytecode files that are out-of-date (the higher-level program has changed but not yet been compiled).

While this approach appears to capture the worst of the other two approaches, there is a different flexibility advantage: a simple interpreter can be provided for each different machine language. This means the same bytecode file can be interpreted (executed) on multiple machines. Bytecode interpreters are both simpler to build and faster to run than high-level language interpreters. Much of the work of a standard interpreter is done in compiling into bytecode so each bytecode executes very quickly.

The hybrid approach is analogous to the use of a particular human language for standard communication within a given field. The Medieval Church used Latin for scripture and communications across Europe because its clergy spoke a wide variety of languages. Rather than having to have an interpreter for every pair of languages (so a Polish speaker could communicate with a Spanish speaker or a Frankish speaker could communicate with a Saxon), the Church could communicate with speakers of any language so long as there was an interpreter from Latin to the given vulgar tongue. Just as the cost of adding a new language to the Church's collection was minimized, the cost of adding a new computer to the list of machines where a given bytecode runs means implementing just a bytecode interpreter.

The hybrid compiler is simpler because it only needs to know one target language, the bytecode. The bytecode serves as the *lingua franca* for multiple CPUs, a language of exchange that they all, through their simple interpreters, can understand. This is very useful when computer programs are shared over the Internet to multiple, different kinds of computers.

Java is a successful example of a bytecode compiled language. In order to run a Java program which we write, we must go through a *write-compile-run* cycle: we must write and save the Java source code; we must provide the Java code to the `javac` Java compiler which translates it into bytecode; we then provide the bytecode file to `java`, the bytecode interpreter on our local machine. One thing to keep in mind is that there is no direct connection between the Java and bytecode files; you must remember to compile the program for any changes to appear in the interpreted program.

Review

- (a) Do the **Xbox 360**, **Playstation 3** and **Wii** have operating systems? Do some internet research to find the answer? Cite your sources when you answer this question.
- (b) Take a look at the **Nintendo 64** game controller. [Insert labeled picture here]. It has two directional controllers - the digital D-pad and the analog stick. How many directions are possible on the digital D-pad? the analog stick? Why is one called analog and the other called digital?
- (c) Is a gaming console a computer? Why or why not?
- (d) Programming languages like C and C++ are compiled into machine code and do not need to be translated as they are executed. Other languages such as Javascript are translated as they are executed (called interpreted). Is the Java programming language compiled, interpreted, or both?

2.3 Java

A Java program is a collection of component types. Each component type, called a `class` in Java, defines the rules followed by components of that type; each is defined in a different file written in the Java language.

Java code is saved in plain text files; this means that they contain simple letters, numbers, and punctuation but not the various formatting constructs saved with a word processing file. Java files must be named for the `class` they contain and must end with the `.java` extension.

The Java language provides for the use of predefined libraries, collections of components that are already written for our use. In addition to the standard Java libraries which Sun ships with their Java development environment, this book uses the Freely Available Networked Game (FANG) Engine. FANG is a library that provides empty games, waiting for you to provide the gameplay.

Syntax

Syntax is defined as “the grammatical rules and structural patterns governing the ordered use of appropriate words and symbols for issuing commands, writing code, etc.” [Dic09] We are interested in Java syntax, how different words and symbols are arranged to make well-formed programs. Notice that well-formed is not the same as correct; a well-formed program has no structural errors but the meaning of the program (known as *semantics*) may or may not mean anything at all, let alone mean what the programmer intended.

A General Class Template

When using a new Java construct, we will summarize the syntax or structure of the construct in a Java template diagram. Our syntax templates use an Extended Backus-Naur Form (EBNF). The form is named for John Backus and Peter Naur and has been in use since the 1950s.

An EBNF notation is a collection of *productions*, that is, rules for creating syntactically correct sequences of symbols. In an application of delegation of work, an EBNF rule or production can refer to other syntactic units by name. In our notation, named units appear between angle brackets. Thus, given the class file defined in Listing 2.1, we can see a template for the `class` construct is:

```
<class> := public class <className>
        extends <parentClassName> {
            <classBody>
        }
```

The production says that the `<class>` construct is defined to be (`:=` is a symbol in the EBNF and reads as “is defined to be”) the word `public` followed by some whitespace, followed by the word `class` followed by some whitespace, followed by a syntactically correct `<className>`.

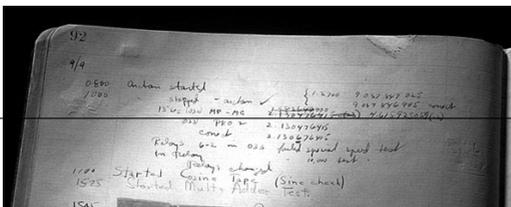
The spaces and line breaks between syntactic symbols indicate that there must be a single space between the elements but that there may be an arbitrary amount of space. The Java programming language is *free-form* in that extra spaces between elements have no meaning, and comments and line breaks are treated as spaces.

The unchanging parts of a template are in the code listing font. The parts which change when you specify a given template are in angle brackets and printed in italics. We have seen an example of a valid `<className>` (`Square`), a valid `<parentClassName>` (`Game`), and a valid `<classBody>` (lines 7-13 in Listing 2.1). All Java programs are composed of some number of classes working together; more details of other, optional, parts of the class template will be presented later. There are also more complicated rules for EBNF diagrams which we will describe when we need them.

High Level and Low Level Languages

Admiral Grace Murray Hopper [1906-1992]

Grace Murray was born in New York, New York, graduating from Vassar in 1928 and receiving her PhD in mathematics from Yale in 1934. While in graduate school she married Vincent Hopper, an English professor.



Grace Murray Hopper (contd.)

Grace Hopper joined the Naval Reserves in 1943, serving on the Harvard Mark I electromechanical computer project as a programmer and operator. Her request to transfer to the active duty Navy at the end of World War II was denied because of her age; she remained in the reserves and worked at Harvard on new generations of computers.

Her work on the Harvard Mark II led to the discovery of the “first computer bug”, an actual moth caught in a relay of the electromechanical computer (see Figure 2.6). She went on to work on the UNIVAC computer line. She was also a member of the team that developed the COBOL

programming language, one of the earliest high-level, compiled programming language.

Grace Hopper retired from the Navy in 1966; she was recalled to the active service in 1967 for a “six month tour” which turned into an indefinite appointment. She retired again in 1971 and was again recalled in 1972; that same year she was promoted to Captain. Before her final retirement from the Navy in 1986 as the oldest active duty officer in the Navy she was promoted to Rear Admiral.

Admiral Grace Hopper spent the rest of her life as a lecturer on computer science around the world. She influenced a large number of computer designers by handing out 30cm (approx. 12 in) wires which she referred to as “nanoseconds” since they were the length that light could travel in one billionth of a second; this was the very farthest that components could be from one another if a computer was to run a billion clock cycles per second.

Two Audiences, One Text

When you write a computer program, you are writing for two audiences. One audience is the computer. You must adhere to the syntax of your programming language and you must use names consistently so that the computer is able to translate from a high-level to a low-level language. The translation program will note errors when it encounters them.

You are also writing for other programmers. You should assume that any program you write will have to be read by someone familiar with the programming language but not completely familiar with the problem you solved and even less familiar with any design decisions you made in creating the program. This is a safe assumption because in a couple of weeks you will find that you are just such a programmer when you come back to your own code.

Comments were mentioned earlier when looking at the `Square` class definition. Comments are one way of providing information for the future human audience of your programs. In Java there are two kinds of comments: end-of-line comments and multi-line comments.

End-of-line comments extend from a double slash, `//`, to the end of the current line. They are treated as if they are not there (the line effectively ends at the double slash). Thus they can go anywhere where a line could be ended.

A multi-line comment is one which begins with a `/*` symbol and extends until the first `*/` symbol. The comment is treated as a sequence of spaces taking up the same number of characters so a multi-line comment can appear anywhere that a space can appear.

The Java template for the two forms of comment is:

```
<comment> := // <commentFromHereToEndOfLine>

<comment> := /*
    <anyNumberOfCommentLines>
    */
```

Since comments are treated as whitespace by the Java language processor (more on the processor in a moment), they are not part of the rules that a Java program follows. Instead, they are there solely for the programmers who come after you. There will be more on writing good comments throughout the remainder of the book. For the moment keep in mind that your comments should document the *design decisions* you have made and guide the reader to how you thought about the program you wrote.

Translating for the Computer

Here we will examine each line of a complete Java program, a completely empty game. It is instructive to look at the differences between the empty game and the square program at the beginning of the chapter. The empty game gives us a chance to examine the parts of a class file.

```

1 import fang.core.Game;
2
3 public class EmptyGame
4     extends Game {
5 }
```

Listing 2.2: An Empty FANG Game EmptyGame.java

Listing 2.2 shows the entire contents of the file EmptyGame.java. The small numbers to the left of the listing are the line numbers. They are not actually part of the listing but are useful for talking about different lines and what they mean.

Our empty game defines a component type, a **class** that has the same name as the file, EmptyGame. That is what line 3 and line 4 together do. Our classes will be declared **public**; this means that the components and their public rules are visible to all Java components⁴.

Each class definition must specify what existing **class** it **extends**; line 4 of EmptyGame specifies that our class extends the FANG class Game. Since Game is a useful, general name, there might be other programmers who wish to use that name. In order for Java to be able to disambiguate among all classes with the name Game, line 1 uses **import** to fully specify that we want the Game class located in the core project which is located in the fang project⁵. By extending another component, each new component *inherits* the attributes and rules of the component it extends.

We use a similar hierarchy of types in chess. All of the movable components are “pieces” regardless of which type of piece they are. Each piece, regardless of rank, has a color. Individual movable components are of type “pawn” or “queen” (and so on) and as such have different movement rules. It is possible, though, to still speak of them generally as in “on your turn, move one of your pieces” (implicitly: using its legal moves). This idea, of hierarchies of types of components, is at the heart of Java’s **class** system.

The end of line 4 and line 5 together contain two characters with which you will become very familiar: { }. These *curly braces* serve as containers in Java. Whenever there is a need to group together sections of code, the code is enclosed inside curly braces. In this case, the empty container contains all the new attributes and rules for our new game.

The empty container means that EmptyGame has *only* the rules and attributes inherited from Game⁶. All together, the definition of this **class** imports a single other **class** definition from the FANG library and then defines a component type that extends that **class** but adds no additional attributes or rules.

Compiling

How do we run this program so that we can see what it does? Remembering that Java uses a hybrid compiled/interpreted model, we must type the file in, save it, compile it, and then interpret it. This book will demonstrate commands you would type on the command-line to run the program; if you are using an integrated development environment like Eclipse, Netbeans, or IntelliJ, you can accomplish the same thing by pressing a couple of buttons⁷.

⁴The rules of visibility, *scope*, are moderately complicated and we defer them for later in the book.

⁵The name fang.core.Game is read “backwards” from right to left, component by component.

⁶We will examine how to find out all the attributes that make up a game in Section 4.4. This section will introduce a few of them as well as other **classes** provided by FANG so we can build *FourOfClubs*.

⁷Take a look at <http://www.fangengine.org/> for step-by-step tutorials in setting up various environments.

Opening a command shell in the folder where we have saved `EmptyGame.java`, we run the Java compiler with the `javac` command:

```
~/Chapter02% javac -classpath ./usr/lib/jvm/fang.jar EmptyGame.java
```

What is `-classpath` and the stuff after it? It is a parameter for the compiler telling it where Java program and library files live. It is a colon-separated list of folder (or directory) names. With two elements, this list tells the Java compiler to look in the current directory and in the `fang.jar` Java archive file⁸. The `fang.jar` archive was downloaded from www.fangengine.org; that website also contains instructions and tutorials on installing the FANG engine.

It is possible for the compiler to detect and report certain types of errors; we will examine some of these in future chapters. For the moment, read the error message very carefully and examine the given line number and the ones just *before* it. A compiler works from the beginning of a file to the end and it may not be able to report an error until it has become completely confused. Thus the reported location of the error is often after the actual location. Make sure that the contents of the file exactly match the code shown in Listing 2.2.

Running

After the `javac` command completes successfully, you will find a new file in the current directory: `EmptyGame.class`. This is the bytecode file generated by the compiler. The Java bytecode interpreter, `java`, is able to start any Java bytecode file which has a special `main` routine; one big thing provided by FANG is a default `main` routine which creates and runs a `Game` (or `Game-derived`) component. The interpreter expects the name of the component to run; it also requires a `-classpath` parameter to be able to find non-standard Java libraries (such as FANG).

```
~/Chapter02% java -classpath ./usr/lib/jvm/fang.jar EmptyGame
```

This should result in the following window appearing on your screen:

In Figure 2.7 the screen is divided into two parts: the top, black portion, the game field itself (empty at the moment because our game is empty) and the row of buttons across the bottom of the screen⁹.

When a FANG `Game-derived` program is run in Java, FANG creates the screen spaces that you see, sets up the game (more on that just below; for the moment set up is empty), and then draws the game on the screen. The game does not actually begin until you press the **Start** button in the lower-left corner of the game window. Pressing **Start** for `EmptyGame` has FANG update the game regularly but, because the update rules for the game are empty, nothing appears to happen.

In fact, when you press **Start**, the only change is the label on the button: it changes from **Start** to **Pause**. This happens whenever you start your game; this permits you to pause the game at any time you like.

The other three buttons permit you to turn on or off the **Sound**, display any **Help** for the current game, and to **Quit** the current game, closing the window completely.

We will now take a moment to discuss the FANG Engine and then look at all the different visual components we can put into the empty game we built here.

Review

(a) What does it mean to compile a Java program? In your answer properly use the terms "source code", "byte code" and "compiler".

(b) Type in the program in Listing 2.1, but leave out line 11. Compile and run the program. Describe what you see that is different and why you think this happens.

⁸In Windows, Linux, and OSX, the two folder names `.` and `..` refer to the current folder (whatever folder you are currently in) and the parent directory of the current directory. By definition, the topmost folder is its own parent.

⁹Due to printing limitations in this book, screen shots will appear primarily in shades of grey; on the screen the top portion is black and the buttons are maroon.



Figure 2.7: EmptyGame

(c) Type in the program in Listing 2.1 but leave out line 12. Compile and run the program. Describe what you see that is different and why you think this happens.

(d) As a new programmer the messages your compiler gives you will be hard to understand at first. One way to start understanding them better is to intentionally put errors in programs and see what the compiler says is wrong. In this way when you encounter the compiler message due to an unintentional error, you have a better chance at figuring out what went wrong. Type in the program in Listing 2.1. Change a part of the program that will make the program have an error when you try to compile it. Try to compile the program and see what message you get. Fix the error, note down what you did to cause the error and write down what message the compiler gives. Repeat this two more times.

2.4 FANG

FANG is the Freely Available Networked Game Engine. FANG is an educational game engine, one designed to support beginning students in being able to write simple computer games early in their careers. This section discusses what an application framework is (game engines are a special kind of application framework) and why you would want to use one when learning Java. We then dissect the FANG name and see why it is a good choice for a beginning programming course.

FANG is the Freely Available Networked Game engine¹⁰. FANG is an example of a *game engine* or an *application framework*. A game engine is software providing the infrastructure for building and integrating with a particular programming language so that programmers can use the features provided by the engine and the associated programming language to build games. An application framework is similar, a collection of infrastructure to simplify building some type of computer application and an associated language so that programmers can build any specific application.

This section quickly covers frameworks and game engines in a little more detail and then takes apart the name

What is a Framework?

An *application framework* is a combination of at least one programming language and a library which provides the infrastructure for building a particular type of application. Application frameworks are typically specialized for use in a particular environment meaning for a particular computer family or operating system. Examples of application frameworks include Cocoa for the Macintosh's OS X operating system, the Mozilla Web browser framework, and the OpenOffice.org framework for building office applications. The important thing to note is that an application framework provides a higher level of abstraction than just a programming language and, as discussed above in relation to high-level languages, computer programmers use abstraction to vanquish complexity.

Large + Flexible = Complex

Java is a large, flexible programming language designed to run on multiple different computer platforms. That is why it uses the hybrid execution model of both compiling and interpreting. It is also why the language ships with close to two thousand different component types in the standard libraries. Java is flexible so that you can build games or instant messaging clients or Web servers or even complete social Web sites containing all three capabilities.

The flexibility of Java and the size of the documentation (we will take a detailed look at using the Java documentation in Section 4.4) make it difficult for beginners (and a fair number of “old hands”) to know where to start when working in the language.

A Java “Tutorial Level”

Many modern video games are also large and complex. Consider *Half-Life 2™* from Valve Software. Just like in its predecessor *Half-Life*, the game begins with several simple levels where a voice in your ear or on the screen guides you through the various movements and attacks that will be necessary to play the game.

Hand-holding during a tutorial level shows new players what to look for as well as familiarizing them with the available controls and the types of problems they will have to solve. A *pedagogical application framework* such as ObjectDraw or FANG is designed as a Java tutorial level: you learn the control structure and how to solve real problems while the framework supports you and lets you reach further than you would have so early without the support.

Freely Available Networked Game Engine

FANG is a collection of several Java *packages* where a package is a collection of component types. The component types provided by FANG will be revealed as the book progresses but packages include the core (where `Game` and `Player` are defined), sound, sprites (visual elements), and networking. You use the packages by specifying the FANG Java archive (`.jar` file) and then `import`ing the components you need.

What does the name *Freely Available Networked Game Engine* mean?

¹⁰For up-to-the minute FANG developments, browse over to <http://www.fangengine.org>. The newest version of the engine, tutorials, and forums are available there.

Freely Available

FANG is an example of *open-source software* (OSS). OSS, also sometimes known as FOSS to include other *free* software, is software where the programmer(s) make the high-level representations of their programs, complete with comments, available to users of the program. This is in contrast to *closed-source* or *proprietary* software where the software vendor makes only the compiled or machine-language version of the software available and typically requires users to agree not to try to reverse engineer the function of the software.

The advantages of OSS are that it is free as in speech and free as in beer¹¹. Free as in beer is obvious: OSS software is made available to anyone who can compile it and, since there are many people all over the world who can compile it, it is typically available even for those who cannot or choose not to compile it themselves. The cost of downloading the source or the binary version is typically the cost of connecting to the Internet. Compare that to the cost of going to a store and purchasing a DVD with the latest version of some particular proprietary program. The cost of free is lower.

Free as in speech is a less obvious benefit but think back to the first computer game you played and really loved. Now, does that computer program run on modern hardware? If all you have is a DVD/CD¹² with the machine code for the game, you have to install it on the right kind of computer (machine code is specific to a particular CPU) and, perhaps, a particular version of an operating system. If you had the *source code* or the higher-level representation of the game, you could compile it for your current machine and operating system.

What if just recompiling the program failed to get it to work for you? The game might be tied to the speed of the processor or some particular type of graphics or networking hardware which is no longer available. With the source code, some knowledge, and a whole lot of patience, you could, at least potentially, fix the game and keep it running. When you have the source code, you can fix any bugs that come along and repurpose the software as you see fit.

Now, imagine if the software was a word processor and that all of your term papers were saved in version 1.0 of the word processor and just before you apply to graduate school version 2.0 of the word processor comes out without the ability to read 1.0 files. After the school upgrades, where are you with being able to put together a portfolio for your graduate school applications? With open source software and formats, you can do what you want with whatever version of the software you want to (as long as you follow the licensing requirements). With proprietary software you have only the options the vendor chose for you.

Networked

Application frameworks provide infrastructure for particular types of applications. One thing they can provide is a communications infrastructure. FANG provides a network infrastructure that makes writing networked games, games with multiple players at multiple locations, almost as easy as writing single player games.

Game Engine

A *game engine* is a special kind of application framework. A game engine provides infrastructure for a particular genre of games. Commercial engines such as Valve's Source Engine or Epic Games' Unreal Engine provide sound, physics, graphics, lighting, networking, and other subsystems to licensees of the software.

FANG was designed as a teaching engine from the get go. Thus it provides well-documented, clear source code. It also provides graphics, networking, and sound subsystems to support the creation of simple, 2-dimensional arcade style games. Because it is open source and flexible, it is possible to write different genres of games but it can be a lot more work than making a game in the genre the engine was built for.

Because FANG was designed as a teaching engine, as we progress through the book, we will peel back some layers of the abstraction provided by FANG and peek "under the hood"; you will get a chance to see how the concepts we use in our games are used inside the game engine as well. You will also be able to, if you wish, extend the game engine. That is the other benefit of open-source software: anyone in the community can contribute back to make the software better.

¹¹The two kinds of free are attributed to Richard Stallman, a OSS pioneer.

¹²Or, heaven forbid, a *floppy disk*.

Structure of a FANG Program

A Java program is a collection of components (implemented as classes and objects of those classes) and rules which govern how they interact. Rules are specified inside of classes, much as the checkers in the moderated version of the game contained their own movement rules. The computer follows the rules which the translator (the compiler) translate into machine code.

The heart of any interactive program, as introduced in Section ?? is the game loop. The loop presented there is incomplete because it only focuses on what happens *during* play. We have implicitly indicated that a game must be setup before it can be played. Using our ability to define rules by giving them names, the following, extended game-playing algorithm captures the whole process more completely:

```
define setup
  create/gather playing components
  set initial configuration

define advance
  update game state

call setup
while (not game over)
  show game state
  get input from user
  call advance

indicate winner (or tie)
```

The new methods defined here, `setup` and `advance` are called in the main program. The `setup` method is called once before the game starts. Once the game starts, `advance` is called over and over to advance the game toward its conclusion. While simplified, this loop matches the internal operation of a FANG game almost exactly. It is up to the programmer defining a game to redefine the empty `setup` and `advance` already provided.

Setting up the Game

Looking back at Listing 2.1, we see a *method* definition for the `setup` method:

```
8  @Override
9  public void setup() {
10     RectangleSprite rectangle = new RectangleSprite(0.5, 0.5);
11     rectangle.setLocation(0.5, 0.5);
12     addSprite(rectangle);
13 }
```

Listing 2.3: Square.java: setup

The first line in the listing, `@Override` tells the Java compiler that this method should redefine an already defined version of the program in the `Game` class. The compiler checks that there is such a method and will throw an error if we say we are overriding and there is no method to override.

The second line is the signature line, the signature of the method being the access level, return type, name, and parameter list of the method. Method definition is covered in detail in Chapter 5. For now, we will just define the four parts of the signature that appear here:

`public` is a Java keyword used to indicate that the thing so labeled is visible and usable outside of the current class. Since the game loop sketched out above runs somewhere inside of FANG, the `setup` method must be publicly available. There are other settings as we will see in future.

Methods can behave like mathematical functions, returning calculated values. Methods returning a value must specify the type of component they return; `void` is a special type indicating that the method does not return any value at all¹³.

The name is next and it must be a single word (no spaces). The next chapter is dedicated to naming things in Java. For now it suffices to know that when overriding or redefining a method we have to match its name and parameter list exactly.

The parameter list is between the parentheses; in this case there is nothing between the parentheses so there are no parameters. Parameters are values passed in to a method to permit it to behave differently each time it is called. Parameters are discussed in detail over the next four chapters.

The *body* of a method comes between the curly braces (opening and closing on lines 9 and 13, respectively). The curly braces serve as a container for the code they enclose. `setup` is called once, before the game begins running, by FANG. When a method is called, the calling method is suspended and the body of the method begins execution with the first line in the body; if no *control statements* change the flow, execution continues in sequence through the body of the method. `Square` declares a local variable, `rectangle`, and assigns a newly created `RectangleSprite` to the variable and then adds the new sprite to the game.

Running the Game

The “games” in this chapter are not really playing. Thus we are satisfied with the empty `advance` method provided by `Game`. Pressing **Start** will start the game loop, just as shown in the algorithm at the beginning of this section. With an empty container for a body, the provided `advance` method does not update anything. So, though the loop is running, there is nothing happening in the game. We will override `advance` in the next chapter when we make our first real game.

Review

- (a) Define the term application framework and give 3 examples.
- (b) Describe the flexibility present in using open source software that is not present when using closed source proprietary software?
- (c) Is it possible to write multiplayer games using the FANG Engine?
- (d) What does the annotation `@Override` indicate?
- (e) What symbols are used to indicate the start and end of a method body?

2.5 An Introduction to Sprites

In computer graphics, a sprite is a two-dimensional visual object that moves in front of the background of the screen. If you consider the game *Pong*, the ball and both paddles could be sprites since they move in front of the ping-pong table background. Historically, *sprite* was coined for an early 1980s video display controller from Texas Instruments [Whi92]. In addition to movement, *sprite* hardware, which was common throughout the 1980s, also supported collision detection (it could tell whether or not two sprites overlapped). In the spirit of that “Golden Age of Gaming” technology, visual game components in FANG extend the `fang.sprite.Sprite` class and have names ending in *Sprite*.

Screen Coordinates

Two-dimensional coordinates are often expressed as a pair of distances: the distance along the *x*-axis from the origin to the coordinate point and the distance along the *y*-axis from the origin. Assuming the two axis

¹³This is the *empty type* or the type with no values, not to be confused with the term `null`, the value meaning *no value provided*. `null` plays an important part in Java and is discussed later.

are orthogonal, each point on the plane has coordinates of the form (x, y) and every pair of numbers refers to a specific point.

Given a point, (x, y) , there are several questions that must be answered to find the point it corresponds to: What is the origin (where is $(0, 0)$)? What is the unit of measure in each dimension (they need not be identical)? What are the unit vectors of the two dimensions (where do the axes point)?

FANG adopts some standards from computer graphics in expressing game locations: The origin is the upper-left corner of the visible game-play area; the x-axis points from left to right (values increase from left to right); the y-axis points down the screen (values increase as you move from top to bottom of the screen).

What units of length are available? It would be possible to use the tiniest *picture elements* or pixels as a unit of measure, except that no two screens are likely to have exactly the same pixel size. It would also be possible, by querying the computer, to figure out what size pixels it has in some more traditional measure of length and then convert all screen distances to millimeters or inches.

Both of these *absolute* measurement systems have a similar problem: the size of the game (and the coordinates of all locations) change when the game window is resized. On modern computers, most windows have “resizing handles” and can, at any time, be resized. Our game would have to know how many millimeters it is from the top of the game area to the bottom of the game area to be able to determine when the apple has fallen to the ground. If the user resizes the game while it is in play, recalculating the size and location of all of the sprites would be difficult.

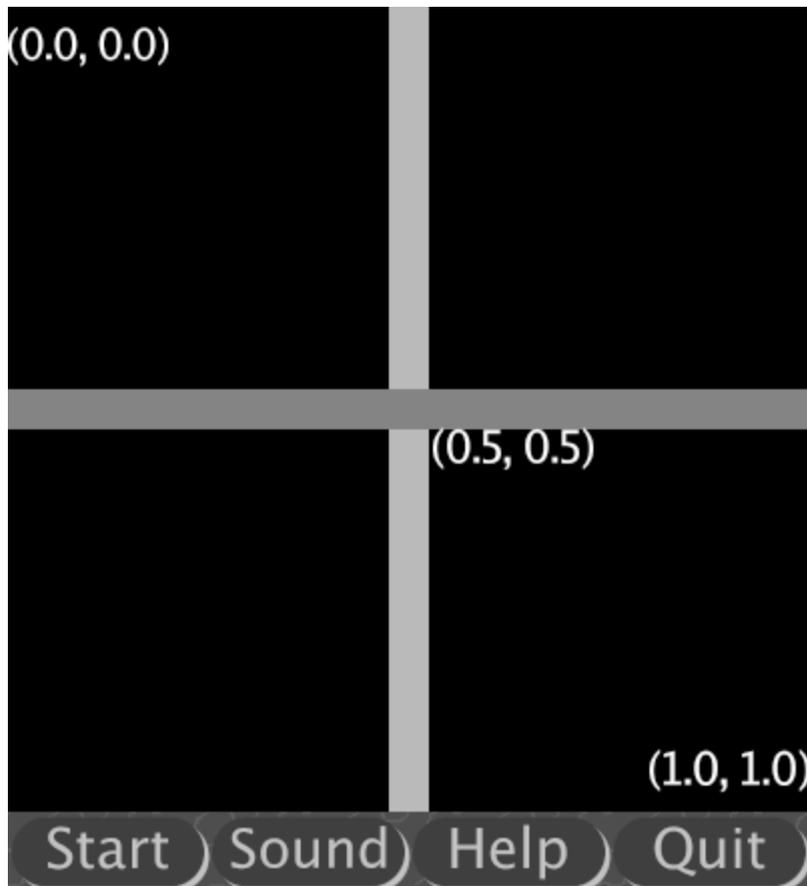


Figure 2.8: FANG Game Coordinates

Instead of absolute units defined outside the game, x and y dimensions are measured *relative* to the current size of the screen. That means that the upper-left corner, the origin, has the coordinates $(0.0, 0.0)$ (the point

is to show that we will be using fractional numbers). The lower-right corner is (1.0, 1.0) (one screen width to the right of the origin and one screen height below the origin).

Figure 2.8 shows the coordinates of the origin and the bottom-right corner. Note that the corner of the game area is just above the top of the bar containing the buttons. The figure also shows a line across the whole game area with the coordinates of the left-most, center, and right-most visible points.

Consider now line 11 in `Square` (Listing 2.1). That line specifies the location of the sprite rectangle to be (0.5, 0.5). That means it is the center of the screen. Since the square is drawn with its center in the center of the game screen, it follows that location of an `RectangleSprite` is the center of the rectangle (this holds for all sprites). Looking at line 10, where the new sprite is constructed, two values, the width of the rectangle in the x and y directions, are provided. These, too, are in screens. Thus (0.5, 0.5) means the rectangle fills a quarter of the game space.

Consider: Which line would you change to make the rectangle one tenth of the screen square? How would you change the line? You could try

```
10 RectangleSprite rectangle = new RectangleSprite(0.1, 0.1);
```

Given that change, what change would move the small square to a spot one third of the screen from the left and one third of the screen from the bottom. The `setLocation` method, line 11, will need different parameters. The first parameter is the x-coordinate, in screens, from the left, and the second parameter is the y-coordinate, in screens, down from the top. We have to convert “one third from the bottom” into a measurement from the top of the screen. One less one third is two thirds. Thus the new line 11 would be:

```
11 rectangle.setLocation(0.33, 0.67); // approximately 1/3 and 2/3
```

We used decimal approximations for the coordinates because Java uses decimal numbers and decimal fractions cannot express one third (or two thirds) exactly.

The coordinates for screen locations are given relative to the current size of the game; this means if the game size is changed, sprite locations change appropriately. The sizes of sprites are also given relative to the screen size so they change appropriately when the game is resized. Note that the location of a sprite is the location of the *center* of the sprite.

RectangleSprite

As we have seen, a `RectangleSprite` can be constructed by specifying the width and height of the rectangle. We define a label for the newly created object (that is what line 10 above does) so that we can use that label to position the sprite, change its color, and add it to the scene.

Now, let's build a simple square “target”: a series of alternating colored squares, centered at the center of the screen, in blue and gold, with each one $\frac{1}{10}$ of the screen smaller than the one around it. With 5 squares, the result should look something like:

Before we go on and write the program, think about how you would specify each of the rectangles: what would its width and height be? What would be the location of each sprite?

Layers and the Order of addSprite

Taking a look at the body of the setup in `Target1.java`, our first attempt at creating a target, we find the following code:

```
18 one.setLocation(0.5, 0.5);
19 addSprite(one);
20
21 RectangleSprite two = new RectangleSprite(0.2, 0.2);
22 two.setColor(getColor("Navy"));
23 two.setLocation(0.5, 0.5);
24 addSprite(two);
25
```

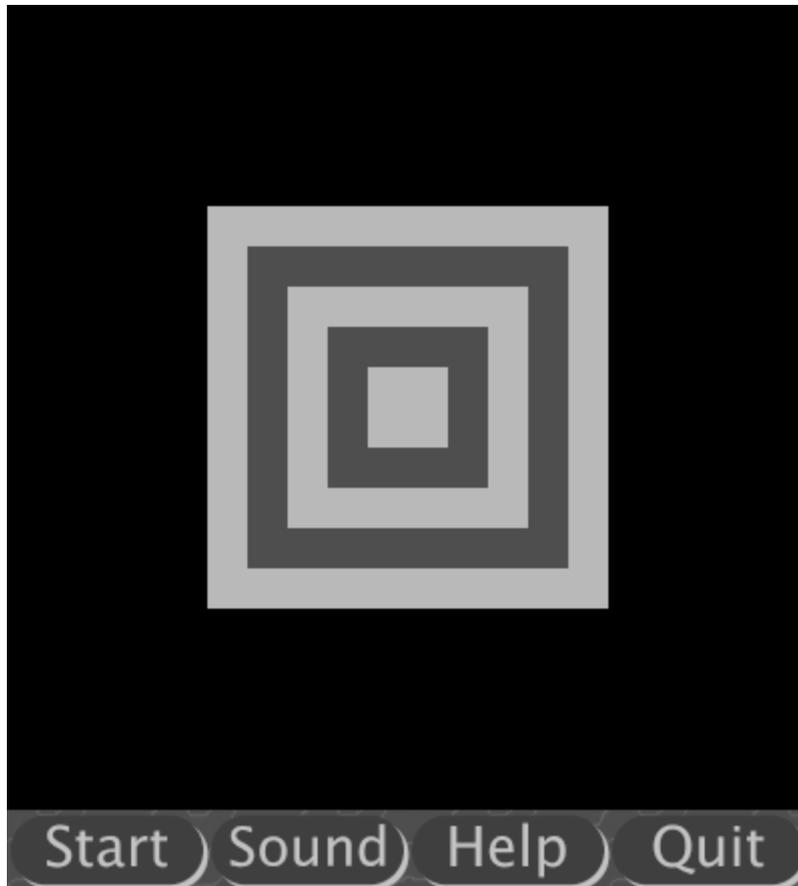


Figure 2.9: Target screenshot

```
26 RectangleSprite three = new RectangleSprite(0.3, 0.3);
27 three.setColor(getColor("Gold"));
28 three.setLocation(0.5, 0.5);
29 addSprite(three);
30
31 RectangleSprite four = new RectangleSprite(0.4, 0.4);
32 four.setColor(getColor("Navy"));
33 four.setLocation(0.5, 0.5);
34 addSprite(four);
35
36 RectangleSprite five = new RectangleSprite(0.5, 0.5);
37 five.setColor(getColor("Gold"));
38 five.setLocation(0.5, 0.5);
39 addSprite(five);
40 }
41 }
```

Listing 2.4: Target1.java: setup body

Five rectangles are created in sizes from 0.1 screen to 0.5 screens across. They are alternately colored navy and gold. They are all centered at the center of the screen. And each, after it is sized, colored, and positioned,

is added to the game. Thus we should see five squares of alternating colors. Yet, when we run the program, the output is as show in Figure 2.10. There is only one gold rectangle visible. What happened to our other rectangles?

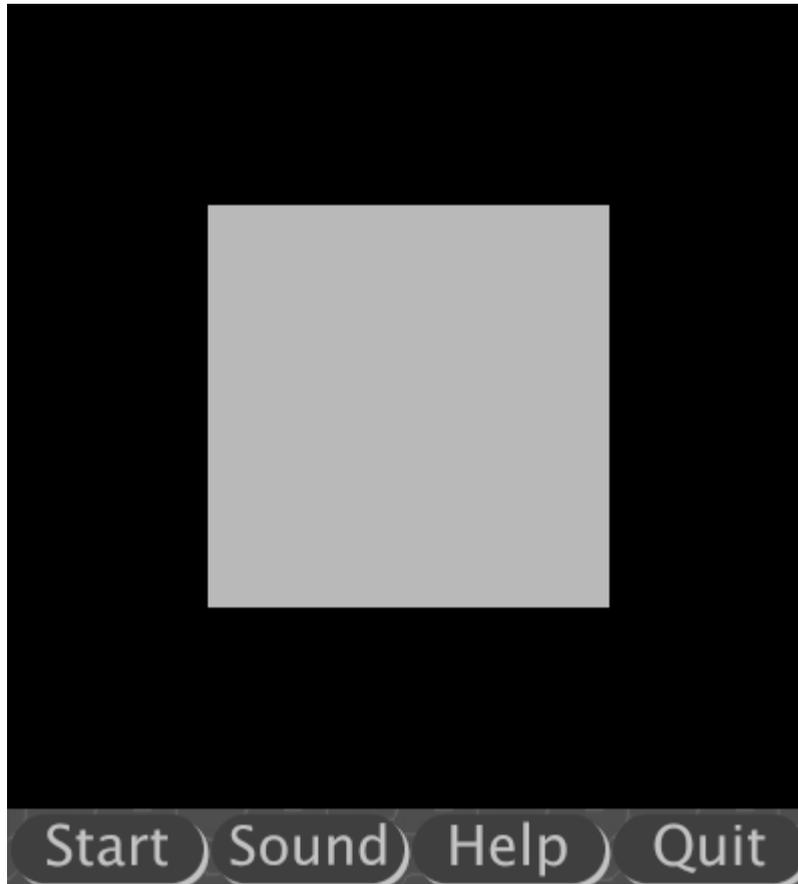


Figure 2.10: Target1 screenshot

While the screen of our game is two dimensional, having an x-axis running from 0.0 to 1.0 from left to right and a y-axis running from 0.0 to 1.0 from top to bottom, there is also a *stacking* or *layering* order. Most drawing programs have this as well. The first visual object drawn on the screen is behind the second visual object which is behind the third and so on. You can think of this as if the sprites were made of construction paper and adding them to the scene was putting them down on the background of the scene. That is, each item is in front of any objects already added to the scene with which it overlaps.

Look back at Listing 2.4. The five rectangles are *added to the scene* in order from the smallest to the largest. The adding to the scene takes place when you call `addSprite`. The order of the other lines is not important. Note that the lines in the body of the `setup` method are executed in sequential order as written. Thus the `addSprite` lines are in the reverse order of the order we need to draw what we want.

Since the names, one through five are arbitrary, we can just reverse the four line blocks which construct, color, position, and add each sprite. The result is `Target.java`, shown below.

```

18
19     RectangleSprite four = new RectangleSprite(0.4, 0.4);
20     four.setColor(getColor("Navy"));
21     four.setLocation(0.5, 0.5);
22     addSprite(four);

```

```

23
24   RectangleSprite three = new RectangleSprite(0.3, 0.3);
25   three.setColor(getColor("Gold"));
26   three.setLocation(0.5, 0.5);
27   addSprite(three);
28
29   RectangleSprite two = new RectangleSprite(0.2, 0.2);
30   two.setColor(getColor("Navy"));
31   two.setLocation(0.5, 0.5);
32   addSprite(two);
33
34   RectangleSprite one = new RectangleSprite(0.1, 0.1);
35   one.setColor(getColor("Gold"));
36   one.setLocation(0.5, 0.5);
37   addSprite(one);
38 }
39 }

```

Listing 2.5: Target.java: setup body

We will see more examples of stacking order and using it to do things we want to do as we draw more pictures.

Line 18 declares a *local variable*. A local variable declaration consists of a type and a variable name. The template for declaring a local variable looks like this:

```
<varDeclaration> := <TypeName> <fieldName>;
```

More Kinds of Sprites

Visual game elements, sprites, all share certain capabilities: setting a location, setting the color, or adding a sprite to the game has the same form regardless of the type of the sprite. The construction of any type of sprite requires two things: importing the appropriate library definition and calling the appropriate constructor.

Looking at the first two lines of Target.java below, notice that each line begins with the keyword **import**; that tells Java to bring in a library class. The name of the library class is then written in reverse order: we want the RectangleSprite class defined in the sprites package which is part of the fang package.

```

1 import fang.core.Game;
2 import fang.sprites.RectangleSprite;

```

Listing 2.6: Target.java: import statements

The Game import is required because we extend the Game class.

Lines almost identical to these lines will come at the start of all of our programs in this section. For each sprite type we will see a program with limited comments, similar to Square.java. Then we'll look at the setup method for another drawing program that does something a little more complicated. Remember that though we will look at lines in the body of setup, in order for the lines to compile you must import the appropriate sprite classes (and have the right -classpath settings).

Round: OvalSprite

An OvalSprite is specified like a RectangleSprite with its width and height. Thus Circle.java makes a centered circle like Square.java

```

1 import fang.core.Game;
2 import fang.sprites.OvalSprite;
3

```

```

4 // FANG Demonstration program: OvalSprite
5 public class Circle
6     extends Game {
7     // Called before game loop: create named oval and add
8     @Override
9     public void setup() {
10        OvalSprite oval = new OvalSprite(0.5, 0.5);
11        oval.setLocation(0.5, 0.5);
12        addSprite(oval);
13    }
14 }

```

Listing 2.7: Circle.java: A complete program

Line 10 declares a name for our new type of sprite, `OvalSprite`, and constructs one half the size of the screen in each dimension. Then the position is set to the center of the screen and the circle is added to the screen.

Because using `OvalSprites` is similar to using `RectangleSprites`, you should consider how easily you could convert `Target.java` into a circular variation.

Can we do anything interesting with layers? What if we had two oval sprites, one white and one black with the black one in front and partially occluding the white one on a black background. That is, what if we ran this setup:

```

17 white.setLocation(0.5, 0.5);
18 addSprite(white);
19
20 OvalSprite black = new OvalSprite(0.7, 0.2);
21 black.setColor(getColor("black"));
22 black.setLocation(0.5, 0.4);
23 addSprite(black);
24
25 OvalSprite left = new OvalSprite(0.2, 0.2);
26 left.setColor(getColor("white"));
27 left.setLocation(0.25, 0.25);
28 addSprite(left);
29
30 OvalSprite right = new OvalSprite(0.2, 0.2);
31 right.setColor(getColor("white"));
32 right.setLocation(0.75, 0.25);
33 addSprite(right);
34 }
35 }

```

Listing 2.8: Ovals.java setup

Figure 2.11 shows the output: `left` and `right` are white circles¹⁴, each centered half-way between the center and the obvious edge of the screen. (a circle is an oval with both diameters the same length). One is centered half way from the center to the left edge, the other half way from the center to the right edge of the screen.

The white oval is wide and short, centered on the whole screen. Thus without `black`, the picture would be an oval centered and two circles above it. What does `black` do to the picture. It is *between* the layers with `white` and the circles. Thus the circles are in front of it (and are not modified). The white oval is occluded by `black` but not entirely. The edge of `black` touches the center of the screen (and thus, the center of the white oval). That means the bottom of the white oval is visible with a curved section cut out of the top.

¹⁴A *circle* is a special case of an *oval*, one where both diameters are the same. An *circle* is to an *oval* as a *square* is to a *rectangle*.

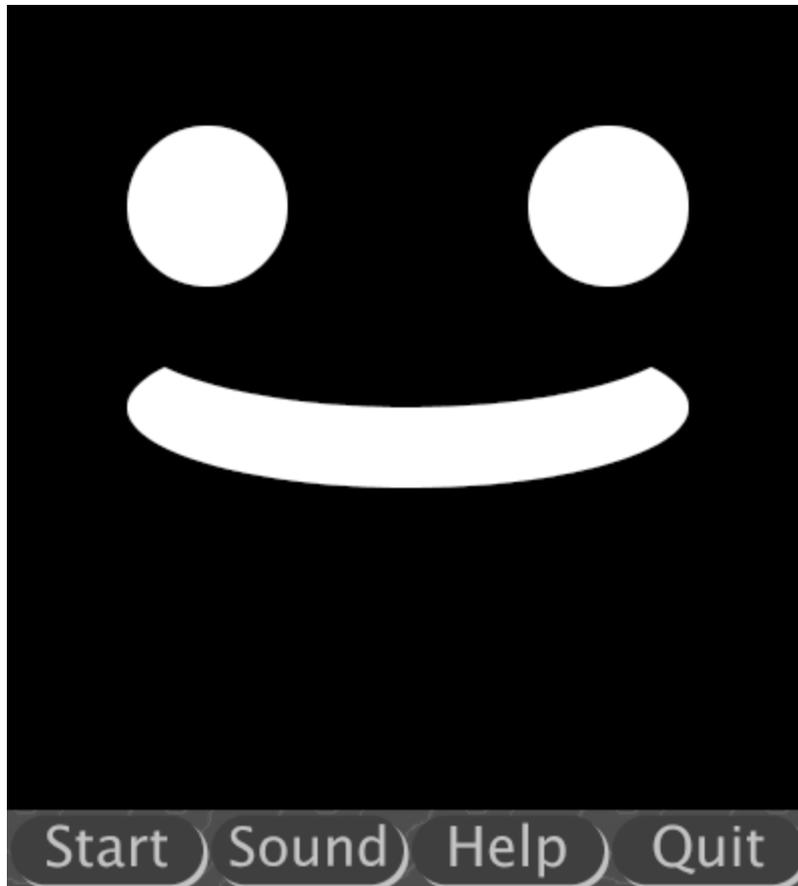


Figure 2.11: Ovals screenshot

One skill you will want to work on is being able to read groups of sprites and figure out what they look like. This permits you to think about making particular figures with combinations of basic sprite shapes, too.

Words: `StringSprite`

The `StringSprite` class is for displaying words, numbers, and anything else composed of characters on the screen. We will use it for displaying things like the score of a game or the number of seconds left in a level. `Hello.java` demonstrates how to create a `StringSprite`.

```
1 import fang.core.Game;
2 import fang.sprites.StringSprite;
3
4 // FANG Demonstration program: StringSprite
5 public class Hello
6     extends Game {
7     // Called before game loop: Draws one string
8     @Override
9     public void setup() {
10         StringSprite string = new StringSprite(0.25);
11         string.setColor(getColor("MistyRose"));
12         string.setLocation(0.5, 0.5);
```

```
13     string.setText("Hello");
14     addSprite(string);
15 }
16 }
```

Listing 2.9: Hello.java: A complete program

Notice first that there is only one parameter passed to this constructor. Again the value is the scale or size of the sprite but in this case we only specify the height. The width of the sprite is determined by the text contents.

That is the second thing to notice: line 13 assigns the text value to the sprite. This is the value it shows on the screen as seen in Figure 2.12.



Figure 2.12: Hello screenshot

As with other sprites, you can create any number of them that you like. In the `setText` method's parameter, you can include the special character sequence `\n` ("slash en") which stands for *new line*. The character will start a new line at the indicated spot in the sprite. This is why we can use four sprites to display six lines in `Strings.java`:

```
18     meddle.setLocation(0.5, 0.15);
19     meddle.setText("Do not meddle in\nthe affairs of");
20     addSprite(meddle);
21 }
```

```

22     StringSprite dragons = new StringSprite(0.15);
23     dragons.setColor(getColor("green"));
24     dragons.setLocation(0.5, 0.4);
25     dragons.setText("DRAGONS");
26     addSprite(dragons);
27
28     StringSprite thou = new StringSprite(0.10);
29     thou.setColor(getColor("white"));
30     thou.setLocation(0.5, 0.65);
31     thou.setText("for thou art crunchy\nand go well with");
32     addSprite(thou);
33
34     StringSprite ketchup = new StringSprite(0.2);
35     ketchup.setColor(getColor("SCG Red")); // red in the book
36     ketchup.setLocation(0.5, 0.85);
37     ketchup.setText("ketchup");
38     addSprite(ketchup);
39 }
40 }

```

Listing 2.10: Strings.java setup

The colors of the four sprites are different, drawing attention to the two colored lines (only one of which really stands out in the text)¹⁵. The text is also emphasized by increasing its size.

Additional features of `StringSprites` will be introduced as we use them (see, in particular, Chapter 9 where we build a game of hangman).

Straight Lines: `LineSprite` and `PolygonSprite`

There are three sprites composed of straight lines. The `LineSprite` class is designed to hold a single line so construction required two end points. The `PolygonSprite` is designed to hold a closed collection of line segments like a triangle or a hexagon; the constructor we will use takes a number of sides and will construct a regular polygon with that number of sides. There is a third variety, the `PolyLineSprite` which can hold an arbitrary number of line segments that are never treated as closed; we will use `PolyLineSprites` in later projects in the book.

```

1 import fang.core.Game;
2 import fang.sprites.LineSprite;
3 import fang.sprites.PolygonSprite;
4
5 // FANG Demonstration program: PolygonSprite and LineSprite
6 public class HexAndLine
7     extends Game {
8     // Called before game loop: make hex and line over it and add
9     @Override
10    public void setup() {
11        PolygonSprite hex = new PolygonSprite(6);
12        hex.setScale(0.5);
13        hex.setLocation(0.5, 0.5);
14        addSprite(hex);
15
16        LineSprite line = new LineSprite(0.1, 0.9, 0.9, 0.1);

```

¹⁵This quote is an homage to JRR Tolkien's "Do not meddle in the affairs of wizards, for they are subtle and quick to anger." [Tol54] The current author was unable to find any authoritative source for the dragon quote.

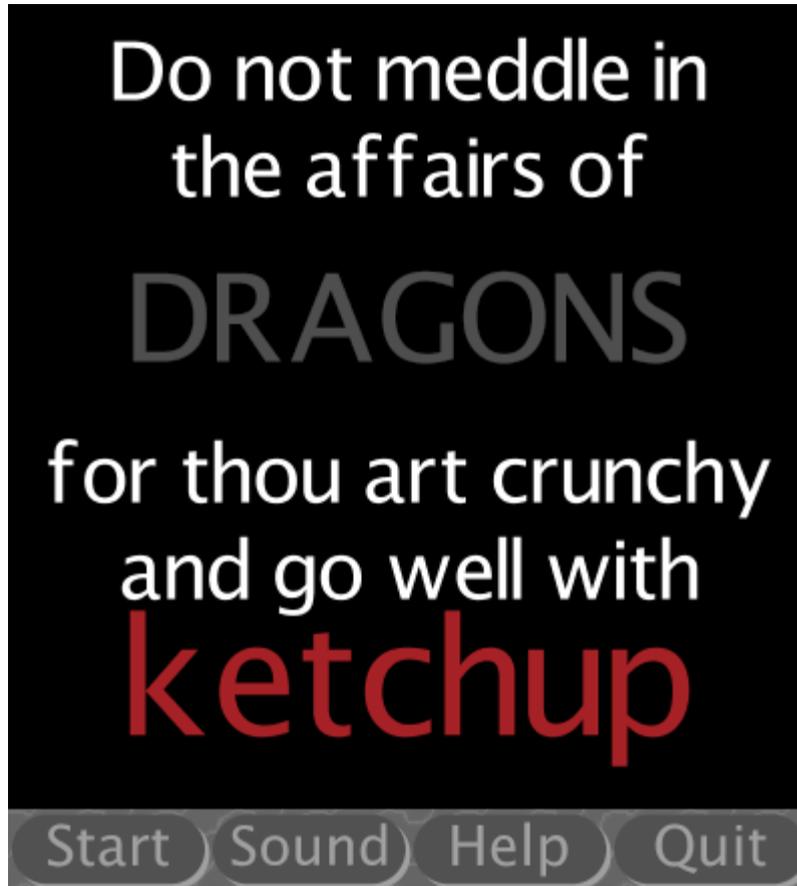


Figure 2.13: Strings screenshot

```

17     line.setColor(getColor("yellow green"));
18     addSprite(line);
19 }
20 }

```

Listing 2.11: HexAndLine.java: A complete program

Our first example, `HexAndLine.java` draws a FANG blue hexagon centered on the screen. Notice line 12 where we set the scale of the hexagon. All sprites support this method and we could have used it to set the size of the squares, ovals, and string sprites we created. The `PolygonSprite` does not have a constructor that takes a scale so we called the `setScale` method to make its size comparable to that of the object in `Square.java` or `Circle.java`.

The `LineSprite` is created and its color is changed from FANG blue (the color you get by default) to yellow green. The next section has more on colors. The line runs at a 45 degree angle down to the left. This is a little discordant in this picture because it is not parallel to any sides of the hexagon.

This section will forgo the more complicated example because specifying the endpoints of a series of corners for a `PolygonSprite` is quite involved. We will come back to these sprites in later chapters.

Pretty Pictures: ImageSprite

Computer images come in two flavors: *vector-based* and *pixel-based* images. Vector-based images are specified in a manner independent of the size of the image on the screen while pixel-based images are specified by supplying the color of each pixel on which the image should be drawn.

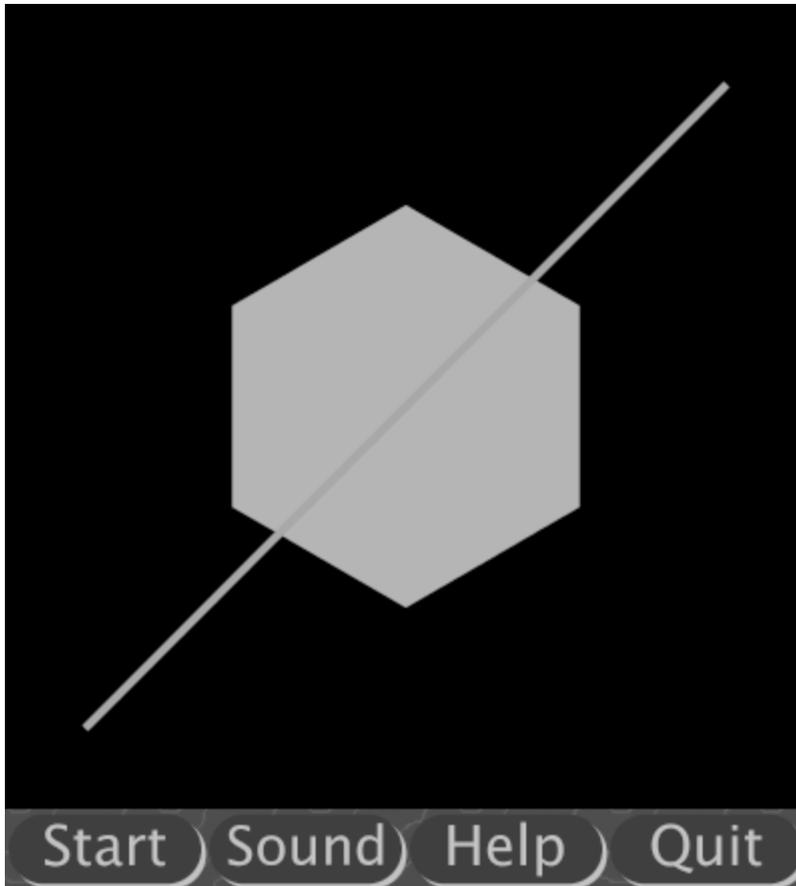


Figure 2.14: HexAndLine screenshot

If you have any experience creating or editing images on the computer, *Photoshop* and the *GIMP* are two of the thousands of photo editor programs available for manipulating pixel-based (or *bit-mapped* images). Digital photographs are examples of pixel-based images. Adobe's *Illustrator* and the open source *Inkscape* are examples of vector-based picture editors. One thing to note is that in most vector-based programs when you add a circle (or other shape) to the scene, it remains a circle, something you can grab and edit later. When you do something comparable in a pixel-based editor, the pixels change color but no scalable object remains.

All of the sprites we have used so far are examples of vector-based objects. After an `OvalSprite` is added to the game, it will resize automatically when the game screen changes size and it will recalculate what pixels are in or out of the circle to keep the edge as clean as possible. We will see later that if we keep a reference to the oval we can change its location, color, or other attributes throughout the game.

FANG also supports pixel-based images, images which were saved into a file with a fixed pixel size (a rectangle of pixels of a given width and height) where the color for each pixel is specified (thus the name *bit-mapped*: the bits specifying each color are mapped onto the rectangle; this is another example of memory contents having meaning determined by the context in which they are interpreted).

The `ImageSprite` class is constructed with the name of a file containing a bit-mapped image (common extensions for such files are `.gif`, `.jpg`, and `.png`). These sprites can be scaled and transformed (as discussed in the next section) but when they are scaled, they often have *artifacts* or ugly, jagged bits, due to the scaling algorithm.

```
1 import fang.core.Game;
2 import fang.sprites.ImageSprite;
```

```
3
4 // FANG Demonstration program: ImageSprite
5 public class Image
6     extends Game {
7     // Load "redJoker.png" in current directory in sprite
8     @Override
9     public void setup() {
10        ImageSprite image = new ImageSprite("./redJoker.png");
11        image.setScale(0.5);
12        image.setLocation(0.5, 0.5);
13        addSprite(image);
14    }
15 }
```

Listing 2.12: Image.java: A complete program

The setup method in Image.java uses a constructor which takes the path to a bit-mapped image file; a path is a sequence of folder names followed by a file name. Here we use the path `./redJoker.png`. This is a playing card image from a set of free card images. The initial dot indicates the folder where the compiled program is being run (the folder with the `.class` file). The result of running this program is show in Figure 2.15. Notice the artifacts around the joker's head.

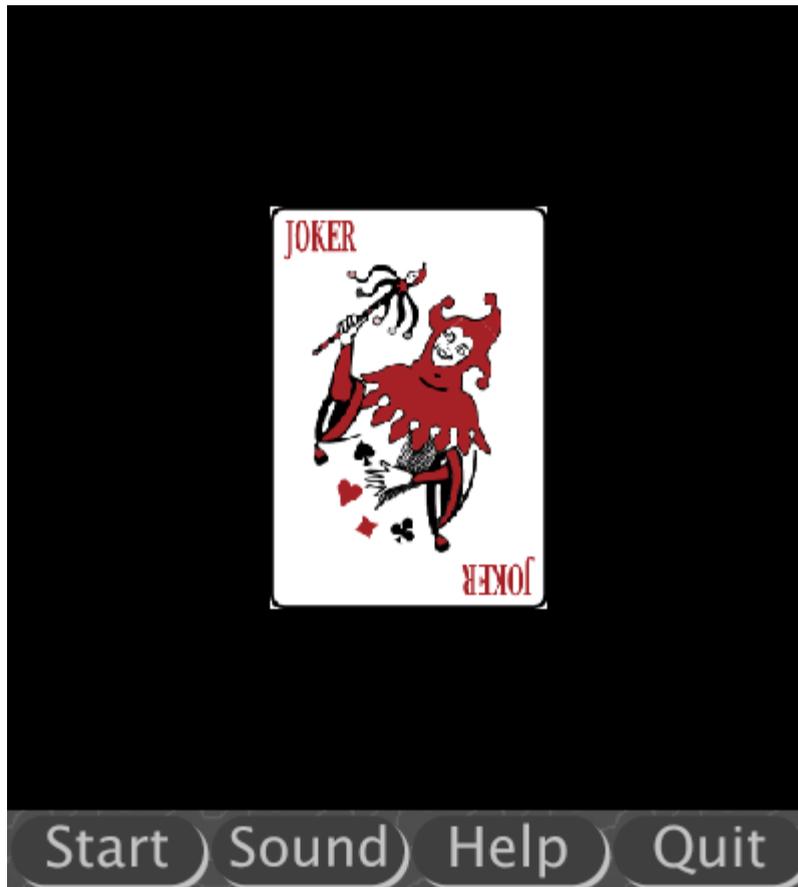


Figure 2.15: Image screenshot

This section introduced most of the sprites in the FANG engine. One good way to figure out what the parameters to the various constructors and methods are is to play with a given program. Try changing the size or the color of various sprites. The next section covers some common sprite methods, methods which all sprites provide. It also discusses color and how to outline a sprite.

More Colors and Transformations

Every time we have used a sprite, after constructing it, we have specified its position on the screen. What is the default location for a sprite? Look at Figure 2.16, a modification of `Target.java` removing all of the `setLocation` method calls.

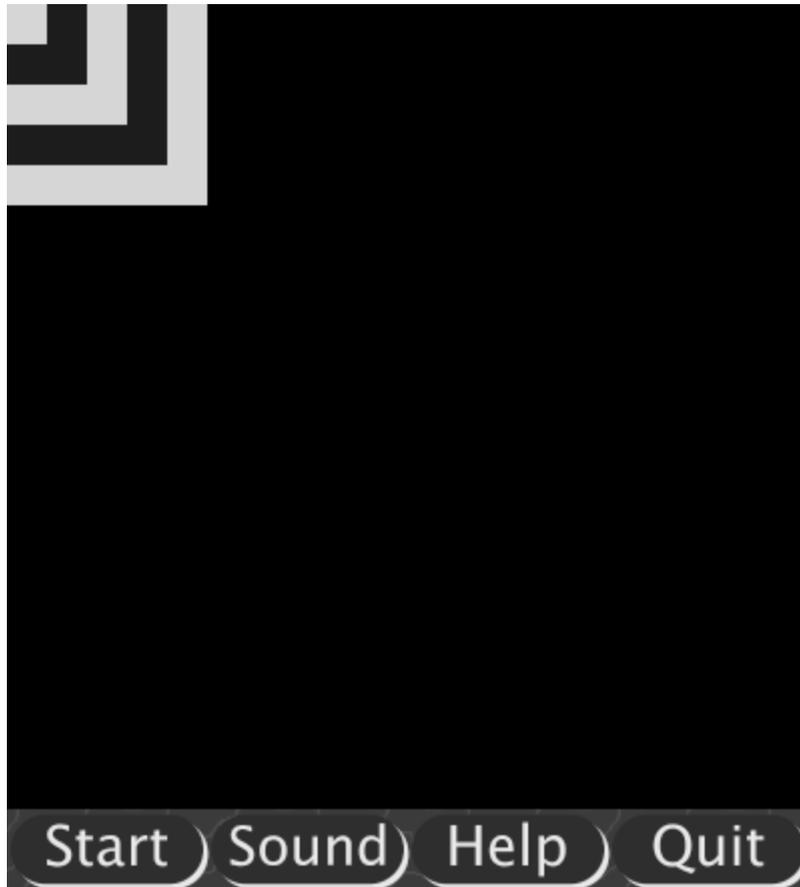


Figure 2.16: Target0 screenshot

The five rectangles are drawn with their centers at the upper-left corner. That is, the default location for any newly created sprite is (0.0, 0.0) or the origin of the coordinate system. Setting the location *translates* or moves the sprite to the specified location. This example also illustrates the fact that the screen you see is just a window onto a larger space and FANG is happy to draw sprites either partially or completely off the screen.

The Java Color Class and FANG

Java provides a `Color` class (you import `java.awt.Color` to use it directly) which specifies color using four *channels*. A channel is either one of the additive primary colors, red, green, or blue, or it is how opaque the color is.

Red, green, and blue: primaries? The computer projects light out of its screen and into your eye (just as a color television set does). Rather than using the subtractive primary colors, red, yellow, and blue, a system projecting rather than reflecting light uses a different set of primary colors. The subtractive primaries are called that because each subtracts some frequencies of light from what it reflects. When you mix all of them together you get black (or, with most paint sets, a dark, muddy brown). The red, green, and blue (RGB) additive primaries each add frequencies to what the screen is projecting and mix together to make white.

We have used different colors for our different examples yet avoided including the `Color` class. That is because the `Game` class has a `getColor` method which returns a `Color`. We have used that method and passed its results to `setColor` for a sprite or `setBackground` for the game.

Colors can be specified by name (see Appendix D for a listing of all color names; the list includes those colors shown for the Web at http://www.w3schools.com/tags/ref_colornames.asp), by specifying the Web numeric value, or by specifying the numeric value for the RGB and, optionally, the opacity channel.

Rotation and Scaling

So far, whenever we create a rectangular sprite, the edges are parallel to the x and y axes of the game. This is not always what we want. Consider drawing the card suit of diamonds. A diamond is a square but rotated an eighth of a circle (or 45 degrees). Just as there is a `setLocation` method, there is also a `setRotationDegrees` method. Thus, to draw a single diamond at the center of the screen, the following setup method will suffice:

```

20
21     // create the diamond, set its color, position, and rotation
22     RectangleSprite diamond = new RectangleSprite(0.1, 0.1);
23     diamond.setColor(getColor("red"));
24     diamond.setLocation(0.5, 0.5);
25     diamond.rotateDegrees(45.0);
26     addSprite(diamond);
27 }
28 }
```

Listing 2.13: Diamond.java setup

Outlining Sprites

One other feature sprites support is outlining their shapes. The following setup method creates two outlined ovals and rotates one of them 45 degrees. The outlines are different colors so we can tell the original from the rotated oval in the screenshot.

```

19     originalWing.setOutlineColor(getColor("white"));
20     originalWing.setOutlineThickness(0.01);
21     originalWing.showOutline();
22     originalWing.setLocation(0.36, 0.6);
23     addSprite(originalWing);
24
25     OvalSprite leftWing = new OvalSprite(0.35, 0.7);
26     leftWing.setColor(getColor("wheat", 128));
27     leftWing.setOutlineColor(getColor("black"));
28     leftWing.setOutlineThickness(0.01);
29     leftWing.showOutline();
30     leftWing.rotateDegrees(+45.0);
31     leftWing.setLocation(0.36, 0.6);
32     addSprite(leftWing);
33 }
34 }
```



Figure 2.17: Diamond screenshot

Listing 2.14: LeftWing.java setup

Lines 23-25 (and 31-33) demonstrate how to turn out outlining: you specify the outline color, the outline size (in screen widths) and, finally, tell the sprite to show the outline. After you specify an outline, scaling the sprite will scale the thickness of the outline as well. This is consistent with the way most vector art programs handle scaling.

The screenshot for `LeftWing.java` shows the original oval, outlined in white, and the rotated oval, outlined in black. This shows that positive rotation is *clockwise*. The screenshot also shows that the color used to fill the ovals is translucent.

Look at line 22. The color is created with the call `getColor("wheat", 128)`. The name, "wheat", is one described in the appendix. The number, 128, specifies how opaque the color should be. If the value were 0, the wheat color would be completely transparent and if it were 255 the wheat color would be completely opaque. The higher the number, the more opaque the color. This means that when we don't specify how opaque to make a color, it is given an opacity of 255.

Note that a sprite rotates around its location. This is why the location of a sprite is typically its center.

Review

(a) Suppose you have a FANG Engine game that is displayed in a 600 wide by 600 tall window and you have a dot in the center of the game. In the FANG Engine coordinate system, what is the x and y location of the dot?

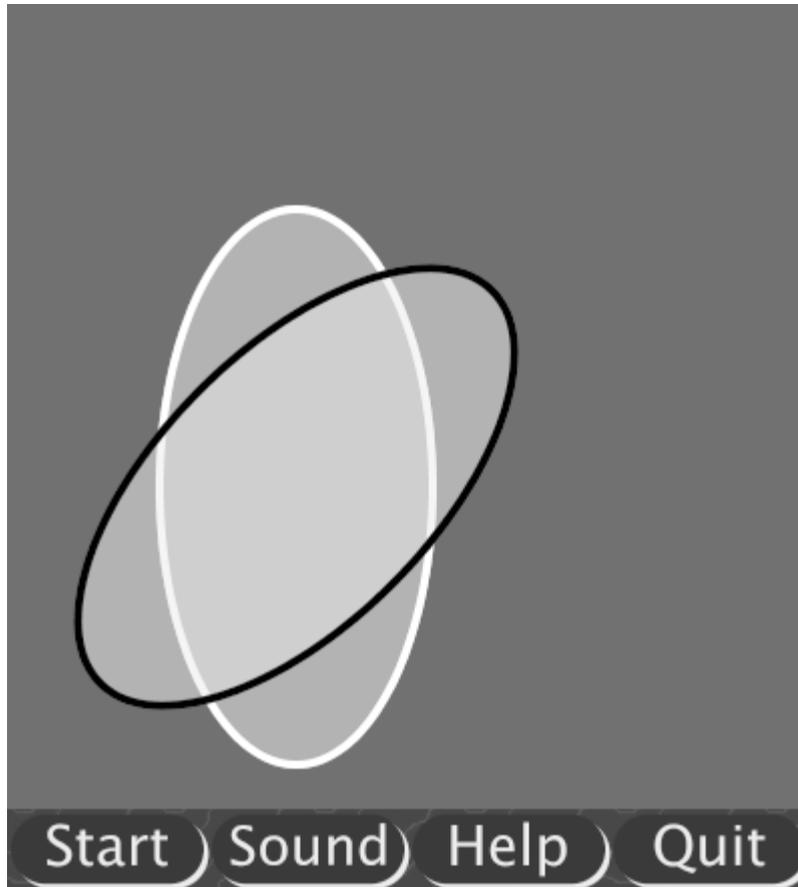


Figure 2.18: LeftWing screenshot

If you resize the window to be 400 wide by 400 tall, would the dot still be in the center of the game window?

(b) Does the order in which you add sprites matter? Why or why not?

(c) Name the sprite you use to create the following:

- (a) a circle
- (b) a square
- (c) some text
- (d) a pentagon
- (e) a diagonal line
- (f) an octagon
- (g) a photograph

(d) Which of the following are NOT valid colors? (Hint: See the appendix on available colors)

- (a) Light Rose
- (b) Rose
- (c) Dark Brown
- (d) Dark Green
- (e) Mauve

- (e) When you use the FANG Engine are you limited to the named colors? Why or why not?
- (f) How do you make a color translucent?

2.6 Get the Picture

This chapter has presented a lot of material with a lot of listings. It has not, yet, finished the proposed program from the beginning of the chapter. The proposed program was to draw a four of clubs. If you go back to Figure 2.1 you will see that a club is composed of three circles and a rectangle.

```

25
26 // Make a black circle a tenth of a screen in diameter just off
27 // center to the left
28 OvalSprite left = new OvalSprite(0.1, 0.1);
29 left.setLocation(0.44, 0.5);
30 left.setColor(getColor("black"));
31 addSprite(left);
32
33 // symmetric position to the right
34 OvalSprite right = new OvalSprite(0.1, 0.1);
35 right.setLocation(0.56, 0.5);
36 right.setColor(getColor("black"));
37 addSprite(right);
38
39 // centered and up
40 OvalSprite top = new OvalSprite(0.1, 0.1);
41 top.setLocation(0.5, 0.43);
42 top.setColor(getColor("black"));
43 addSprite(top);
44
45 // stem is centered
46 RectangleSprite stem = new RectangleSprite(0.04, 0.1);
47 stem.setLocation(0.5, 0.5);
48 stem.setColor(getColor("black"));
49 addSprite(stem);
50 }
51 }

```

Listing 2.15: Club.java setup

The setup method in Club.java does just what you would expect: it sets the screen color to white (so the black clubs show up) and then constructs and adds to the game three circles and one thin, vertical rectangle.

Figure 2.19 shows the results of running Club. How could we modify this program to create the four of clubs we originally designed?

One answer is to copy the twenty or so lines which actually draw the club four times and modifying the exact location in each of the copies. Creating that program is left as an exercise for the reader. It is not a good approach because smart computer scientists do not repeat themselves.

The Do not Repeat Yourself Principle. (DRY) Whenever possible, you should not repeat yourself when writing a computer program. As you learn to use iteration and delegation, you will find that when you need to do the same thing over and over, with little changes, it is always worthwhile to define a new command that takes the little changes as parameters. The DRY is one of the simplest rules of *software engineering*, the discipline studying how to manage writing high-quality code, yet it is one of the most effective¹⁶.

¹⁶The current author first encountered the DRY Principle in this form while reading Hunt and Thomas' *The Pragmatic Programmer* [HT99]



Figure 2.19: Running Club

Why don't you want to repeat yourself? What if you forget to change one of the copies? Then you have what looks like 3 clubs and you have no idea where the "extra" one is. Worse, what if you get it to work and then, in a week, want to change it and don't remember which parts are copies of the drawing code? And even worse, what if you need to fix how clubs draw because you want to use a more pleasing stem made from a `PolygonSprite`. Which sprites do you change?

What we want to do is create another method.

Methods: New Rules

When we first wrote a `setup` method, we discussed the signature of a method: the access level, return type, name, and parameter list. We then discussed that the body of the method executes from beginning to end in sequence (until we see how to handle selection and iteration). All of that will help us write a new command, a second method in one class. The method, called `club` is listed below.

```

44     stem.setLocation(centerX, centerY);
45     stem.setColor(getColor("black"));
46     addSprite(stem);
47 }
48
49 /**
50  * Setup the game. This method is called once automatically by FANG

```

```

51  * before the game begins running. This is where the components for
52  * the game are built. This "game" is really a picture of the card
53  * symbol "club".
54  */
55  @Override
56  public void setup() {
57      // make the background white (so a black club will show)
58      this.setBackground(getColor("white"));
59
60      // draw four clubs
61      club(0.25, 0.25);
62      club(0.25, 0.75);
63      club(0.75, 0.25);
64      club(0.75, 0.75);
65  }
66  }

```

Listing 2.16: FourOfClubs.java club

The first thing to notice is how similar this code is to setup in Listing 2.15. In fact, it was cut from the setup method and then generalized so the club could be centered anywhere on the screen. Whatever values centerX and centerY have determines where the center of the club is.

That means that in line 48 (and all other setLocation lines) the center of the new sprite is determined relative to the values of centerX and centerY. Thus, since the first club was drawn at the center of the screen *and* the location of the left leaf was (0.44, 0.5) (see Listing 2.15, line 31), the position, relative to the center of that club, is (-0.06, 0.0). That is, the x-component of the center of the left leaf is 0.06 to the left of center and on line with the center vertically. That is exactly what the two expressions in setLocation in line 48 do.

The other three sprites are positioned relative to the center of the club as well. So, now, we need to figure out how centerX and centerY get their values.

Line 44 is the signature of club. The method is **public**, it returns nothing, hence the return type of **void** and its name is club. These three parts of the signature match the signature we use for setup in all FANG programs. The only difference is in the parameter list (okay, an in the *actual* name of the method).

The parameter list is a comma-separated list of type-name pairs. That is: **double** centerX, **double** centerY is a list of two such pairs. That means that when calling the method we must provide two numbers (**double** is a numeric type in Java; lots more on this in the next two chapters). We will list the two numbers between parentheses when we call the method and we will separate them with a comma.

```

23  // Make a black circle a tenth of a screen in diameter just off
24  // center to the left
25  OvalSprite left = new OvalSprite(0.1, 0.1);
26  left.setLocation(centerX - 0.06, centerY);
27  left.setColor(getColor("black"));
28  addSprite(left);
29
30  // symmetric position to the right
31  OvalSprite right = new OvalSprite(0.1, 0.1);
32  right.setLocation(centerX + 0.06, centerY);

```

Listing 2.17: FourOfClubs.java setup

The setup method of FourOfClubs, Listing 2.17, shows how to call our new method. club(0.25, 0.75) calls the method with centerX set to 0.25 and centerY set to 0.75. That draws a club centered a quarter from the left and a quarter from the bottom of the screen.

It is likely that you can follow the discussion of how club works but are not yet sure how to write a method of your own. This is to be expected. Make sure you understand how *this* method works and, in particular, how the parameters, centerX and centerY get their values.

Review

- (a) Why is it not a good idea to have very similar code repeated in your program?
- (b) How do methods help you apply the principle of Do not Repeat Yourself (DRY)?
- (c) What is the difference between calling a method and defining a method?

2.7 Summary

How Computers Work

Modern computers are *digital* (having discrete rather than continuous levels) and *binary* (working in base 2; values limited to 0 and 1). A single *bit* (binary digit) can hold two different values; by combining bits into larger groups, arbitrarily complex values can be encoded.

Encoding is an important concept: The *meaning* of the contents of computer memory depends on the *context*. The same computer memory can hold instructions, whole numbers, floating point numbers, text, music, or images. In fact, any given pattern in memory might be interpreted as any of those types. This is the power of computers, that something that is data to one program can be treated as instructions by another.

Levels of Abstraction and Languages

Computer program design requires working at multiple *levels of abstraction*. This idea of levels applies to programming languages: *high-level languages* are general and separated from the hardware level while *low-level languages* are typically closer to the hardware.

High-level languages have *source code*, the programs expressed in the high-level language, which can be *compiled* into a lower-level representation of the same program. The *compiler* translates the whole program at once. An *interpreter* runs the higher-level program more directly, interpreting each line and running the interpreter's routines to do what the line says.

Java is a high-level language which is compiled into *bytecode*, a machine language for a virtual computer. The bytecode is then interpreted so that the same bytecode can run on multiple computers without change.

Working from high to low levels of abstraction also works in designing programs, solving a given problem by expressing a solution as a combination of solutions to simpler problems. This same approach can then be applied at different levels as well.

Sprites

Sprites, named for pixies who can fly around, are FANG screen objects which can be located and transformed on the computer screen.

General Properties and Methods

All FANG sprites support a common set of methods which include:

<code>setColor(color)</code>	Set the fill color of the sprite to the given color.
<code>setScale(scale)</code>	Set the scale of the sprite in screens.
<code>setLocation(x, y)</code>	Move the center of the sprite to the location (x, y) where the values are in screens.
<code>setRotationDegrees(degrees)</code>	Rotate the sprite <code>degrees</code> degrees clockwise.

The Game class has a couple of methods for working with colors and sprites, too:

<code>getColor("color name")</code>	Get the named color. Will cause run-time error if named color does not exist.
<code>getColor("color name", opacity)</code>	Get the named color but set the opacity to the given value. <code>opacity</code> must be an integer on the range[0-255].
<code>addSprite(sprite)</code>	Adds the <code>sprite</code> to the game at the top of the current stacking order.

Constructors for Different Types of Sprites

We saw one constructor for each of several types of sprites. The constructors, appearing after the `new` operator, construct a new sprite of the given variety.

```
RectangleSprite(width, height)
OvalSprite(width, height)
StringSprite(height)
PolygonSprite(numberOfSides)
LineSprite(x1, y1, x2, y2)
ImageSprite("image file path")
```

All heights, widths, and screen coordinates are given in screen units.

FANG Classes and Methods

Java Templates

```
<class> := public class <className>
        extends <parentClassName> {
            <classBody>
        }
```

```
<comment> := // <commentFromHereToEndOfLine>
```

```
<comment> := /*
            <anyNumberOfCommentLines>
            */
```

```
<varDeclaration> := <typeName> <fieldName>;
```

Chapter Review Exercises

Review Exercise 2.1 Describe the function of each of the following parts of a modern computer

- Central Processing Unit (CPU)
- Random Access Memory (RAM)
- Hard Disk/Solid-state Memory
- Keyboard/Game-Controller
- Screen/Printer

Review Exercise 2.2 What does *volatile* mean? How does it apply to computer memory?

Review Exercise 2.3 Why do hard drive manufacturers insist that KB, MB, and GB refer to 1,000, 1,000,000, and 1,000,000,000 bytes respectively, rather than the values which computer scientists prefer?

Review Exercise 2.4 Modern machines are digital, binary computers.

- What does digital mean?
- What does binary mean?

Review Exercise 2.5 Can you give a real-world example where the contents of some storage are interpreted differently depending on the context? That is, where the expected *type* of the content colors the meaning it is given?

Review Exercise 2.6 What is the difference between *high-level* and *low-level* programming languages?

Review Exercise 2.7 What is a *compiler*?

Review Exercise 2.8 What is an *interpreter*?

Review Exercise 2.9 Define the following Java language terms:

- (a) `class`
- (b) `extends`
- (c) `import`

Review Exercise 2.10 What is a *sprite* in FANG?

Review Exercise 2.11 When does FANG call each of the following methods (if you define one):

- (a) `public void setup()`
- (b) `public void advance(double secondsSinceLastCall)`

Review Exercise 2.12 What does the compiler do with comments? For whom are comments written?

Review Exercise 2.13 Explain why it is important to “document your intent” in comments rather than just describing the Java code.

Review Exercise 2.14 What is the screen coordinate for each of the following:

- (a) the upper-right corner of the game screen?
- (b) the middle of the left side of the game screen?
- (c) the center of the screen?

Review Exercise 2.15 Where on the screen (what set of points) are the screen coordinates the same? That is, where are the x-coordinate and the y-coordinate equal?

Review Exercise 2.16 What are the screen coordinates of a point one-third from the left edge and two-thirds from the top of the screen?

Review Exercise 2.17 Start with `Oval.java`. Modify the background color of the game to “yellow green”.

Review Exercise 2.18 How many different named shades of grey are available in FANG?

Review Exercise 2.19 Start with `Target.java`. Modify the program to display a circular target rather than a square target.

Review Exercise 2.20 Start with the circular target from the previous question. Modify it so that the alternating colors are red and white.

Review Exercise 2.21 Start with `Hello.java`. Modify the program to display your first name centered across the top of the screen.

Review Exercise 2.22 Start with the first name program from the previous question. Add your last name in a contrasting color, centered along the bottom edge of the screen.

Review Exercise 2.23 Start with `Target.java`. Modify the program to display a hexagonal target.

Review Exercise 2.24 What command would you use to create a `LineSprite` that ran from the upper-right corner down to the lower-left corner of the screen?

Review Exercise 2.25 Start with `Ovals.java`. Modify the program to add “eyes” to the drawing.

Review Exercise 2.26 Draw a design diagram for a picture of a flamingo. Make the picture as simple as possible using simple geometric shapes.

Review Exercise 2.27 What are the five problem solving techniques described in the text?

Programming Problems

Programming Problem 2.1 Given the Figure 2.20 design drawing

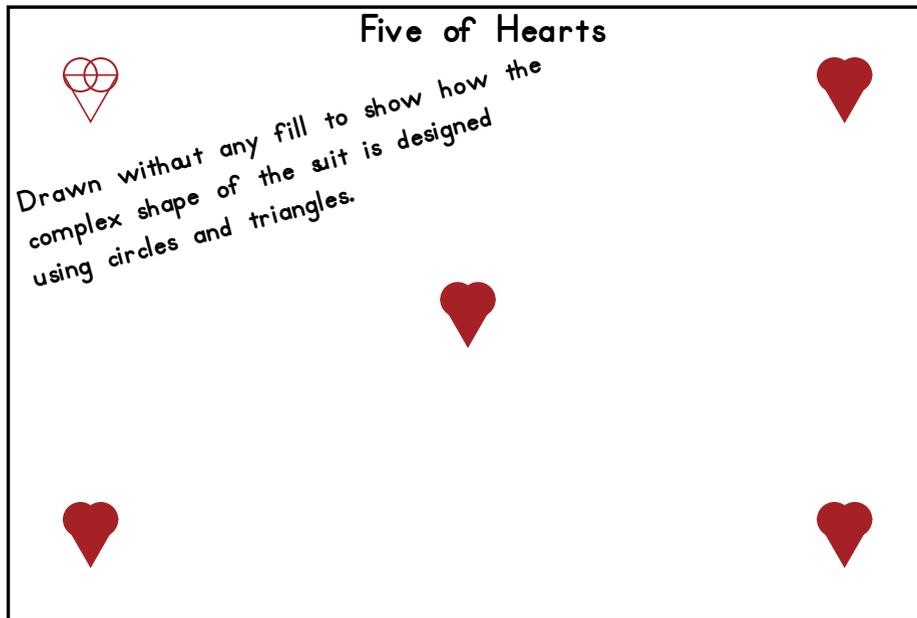


Figure 2.20: FiveOfHearts game design diagram

- Write a program, `Heart.java`, which draws a single heart in the center of the screen.
- Modify `Heart.java`, creating a method, `heart`, which draws a heart at a given location on the screen. Then, using `heart`, draw a five of hearts as shown in the design diagram. The signature of `heart` is shown below.

```
public void heart(double centerX, double centerY) {
    ...
}
```

Programming Problem 2.2 Given the Figure 2.21 design drawing

- Write a program, `Spade.java`, which draws a single spade in the center of the screen.
- Modify `Spade.java`, creating a method, `spade`, which draws a spade at a given location on the screen. Using `spade`, draw a three of spades as shown in the design diagram. The signature of `spade` is shown below.

```
public void spade(double centerX, double centerY) {
    ...
}
```

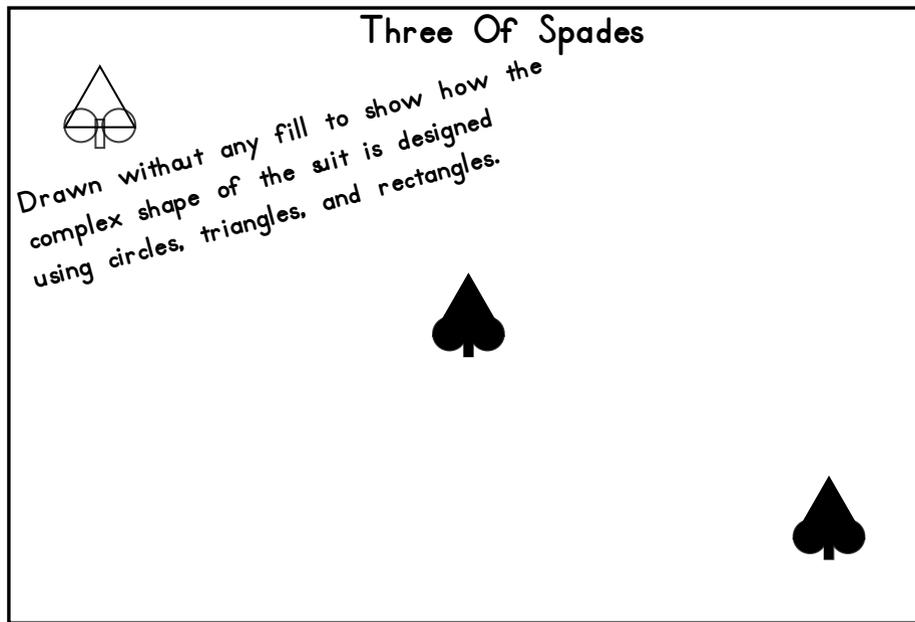


Figure 2.21: ThreeOfSpades game design diagram

Programming Problem 2.3 Given the screenshot in Figure 2.22, write a program to draw a bee. Be sure to use at least three colors. Also note that the wings are partially *transparent*.

Programming Problem 2.4 Write a program to draw a multi-color clown face. Use at least three different shapes and at least three different colors.

Programming Problem 2.5 Write a program to draw a house. Your house should have at least two windows and at least three colors.

Programming Problem 2.6 Write a program to draw a top-down view of an airplane. The airplane should be as simple as possible so that it can be recognized at fairly small resolutions.

Programming Problem 2.7 Start with `Ovals.java`. Modify the program so that instead of just drawing the face once, it has a `face` method which will draw a face at some particular location on the screen. Then, using `face`, draw four faces on the screen.

Programming Problem 2.8 Start with `Ovals.java`. Modify the program so the face is drawn tilted 45 degrees clockwise.

Programming Problem 2.9 Write a program that changes the background of the game to a light color and uses simple geometric shapes to draw a living broccoli stalk. The broccoli should have a face and be as large as possible given the screen space.

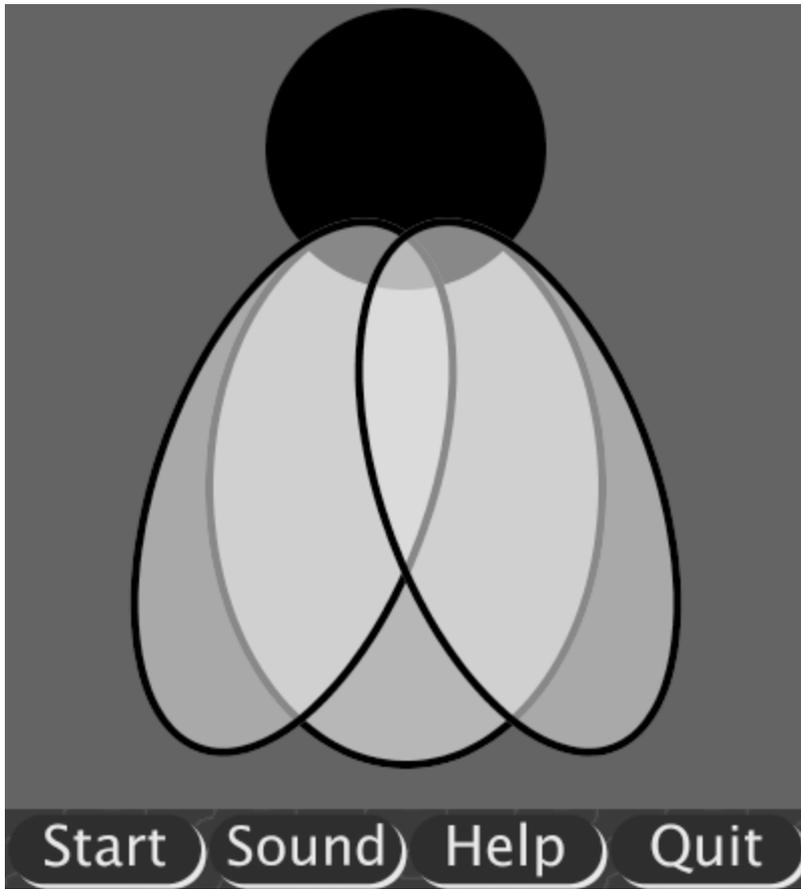


Figure 2.22: Bee screenshot

Deciding What Happens: `if`

Chapter 1 defined a game as a collection of components and associated rules governing their interaction. The definition also included the stipulation that the game presents the user with meaningful choices that shape the outcome of the game.

Chapter 2 introduced Java, presented several sample programs using sprites, and described how the video game loop is at the center of all FANG programs.

To define our own game, we *override* the two methods `setup` and `advance`. The one thing conspicuously missing from the previous chapter is giving the user any choices at all. The programs all simply draw a picture. One reason for that is that introducing selection, choosing one or another set of rules depending on some condition, seemed like overkill with all the syntax juggling in Chapter 2.

This chapter will start by designing a simple game, as simple a game as possible. We will introduce a little more terminology around game design and then work on translating the game into the two methods we normally override in `Game`. Along the way we will reexamine the list of problem solving techniques and look at sequence and selection in detail.

3.1 A Simplest Game

By focusing on *simple* computer games, this book teaches traditional computer science topics. For our first real game, we will go further than looking at simple games to looking at a simplest game. This chapter then develops a complete game design for a simplest game and then on translating that game design into a FANG program.

Why say “a simplest game” rather than “*the* simplest game”? The short answer is that there are any number of games with a minimal number of components and rules, different outcomes, and choices the player makes that influence those outcomes. Some different simplest games are suggested in the programming exercises at the end of the chapter.

Fewest Components and One Rule Follower

Simplicity dictates that we minimize the number of things in the game. That is, we want the minimum number of players interacting with the minimum number of components, each with a minimum number of rules.

A single-player game has the fewest players possible. The computer controls the environment within the game and any opponents; the computer provides the rule followers for any elements of the environment or opponents requiring them.

Computerized opponents require strategies; the program must have rules for both the *game* and *winning* the game. This violates our current goal of simplicity. A computer game where the computer enforces the rules is a form of solitaire. Note that many arcade games from the 1980s and 1990s (the Golden Age of video games to many) are solitaire games.

The minimum number of components is also one; it is difficult to imagine rules where a single component on the screen constitutes what we would consider a game. The fewest components in a viable *game* is two.

Now for the rules. We will consider games where the player directly interacts with at least one of the components. Thus one component in some way represents the player. The other component and the screen represent the “rest” of the game: the state of the game is communicated to the user through the single remaining component.

Many games are possible with just these simple components. Gameplay depends on many things: the computer component could move or remain stationary; the user could control the position, facing, speed, or size of their component; when moving, components might interact with each other or the edges of the screen in different ways.

If the computer component moves on its own, the player can be tasked either with catching it or avoiding it. Our first game, *NewtonsApple*, will have the player catching the computer-controlled component.

In *NewtonsApple*, the player plays (moves) a component representing the famous physicist, Sir Issac Newton. The computer will randomly “drop” apples by placing them along the top of the screen and moving them toward the bottom. While limited to moving from side-to-side, the player must move Sir Issac so that each apple falls on his head. *NewtonsApple*’s score will be the ratio of the number of apples caught to the total number of apples dropped. Figure 3.1 draws a picture of the game play area.

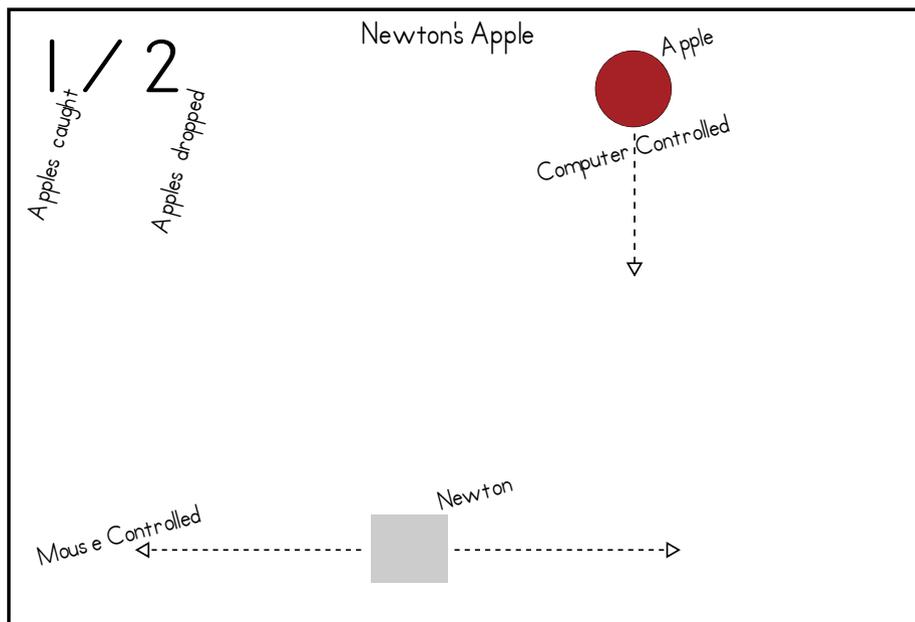


Figure 3.1: NewtonsApple game design diagram

There is a common sense dictum that when you don’t understand some problem, draw a picture. This holds particularly true in game and computer program design. This sort of diagram, a picture of the screen with components and some notes on their behavior, captures many of our design decisions more clearly and compactly than natural language. The design task is explicitly about taking a general idea and specifying it, understanding it well enough to be able to make the game or program. Deep understanding is also required to permit us to imagine how the game could be changed to make it more fun or how to translate it into a different medium.

Chapter 1 discussed the idea of differentiable outcomes, winning or losing. Differentiable outcomes imply some way of keeping score. In the text description of the game this was mentioned as “the ratio of the number of apples caught to the total number of apples dropped.” Our diagrams help us realize when important things are *missing*: we have added a third component, the score. Remember that when designing a game or a program, it is important to make everything explicit.

Scores are an important part of a game; players use them as feedback on how well they are doing and can compare them with other players for an indication of relative abilities. Our score tracks “trials” and “successes”. This means that whenever an apple stops falling (it is caught or it hits the ground), the number of trials is incremented; whenever an apple is caught, the number of successes is incremented. We will use different scoring metrics in other chapters.

FANG and the Video Game Loop

The default program structure for the Game class is based around a video game loop and two *extension points*. An extension point is a method, typically provided in a framework, which the framework designer expects users to override. The built-in version of the method may provide some sort of minimal function or, as in FANG, no function at all. The core game loop, with extension points is as follows:

```
define setup
  do nothing

define advance
  do nothing

setup
while (not game over)
  displayGameState
  getUserInput
  advance
```

The setup method is overridden with code to set up the game. That method was extensively explored in the previous chapter.

displayGameState is where FANG goes through the list of all sprites that have been added to the game and draws them on the screen¹. The getUserInput step is done to see what has happened with the mouse or keyboard. The advance step is where, based on the current game state and the user’s input, the new state of the game is calculated. Then the whole loop begins over again with displaying the new game state.

The *game state* is the current condition of the game. You could consider the game state for chess to be the current configuration of the board, the time on the chess clocks for each player, and, possibly, the list of moves that got us to this point. The state of a game of tennis is the score, who hit the ball last, the location and “state” of each player (tired, thirsty, distracted, etc.), and the location and state of the ball (velocity, bounciness, etc.).

The advance Method

The signature of the advance method is

```
public void advance(double secondsSinceLastCall)
```

It is **public** so the FANG engine can call it. It returns **void** (or nothing at all). It is called advance and it has a single element in its parameter list. That value is a **double** or fractional number (more on numbers in the next two chapters). FANG tries to advance about twenty times a second so secondsSinceLastCall typically has a value near 0.05. The exact value is passed in so that it can be used for calculating things like how far the apple has fallen since the last time it was moved.

¹FANG uses the *painter’s algorithm*, drawing each sprite in order from the back to the front; this is why sprites added later occlude sprites added earlier. Note that it is possible to reset the ordering of the sprites.

Overriding

We discuss *overriding* the advance and setup methods. What does override mean? In object-oriented programming languages such as Java, classes can extend other classes². When the child class, the one extending the parent class, has a method with exactly the same signature as a method in the parent class, the child's method is said to override the parent's version.

This is important because when Java calls a method on an object of a class, if there are overridden methods, it will always choose the *lowest* definition of the method. That is, no matter where a reference to a Game is used to call advance, even inside of FANG's video game loop, because we override advance in NewtonsApple, the NewtonsApple version will be called.

Overriding is how extension points are implemented in object-oriented programming languages.

Before we fill in the body of the advance method, we will take a short time out to discuss how programs (and games) are designed and how problems are solved.

Review

- (a) According to our definition, is *Newton's Apple* a game? Why or why not?
- (b) Two methods of the Game class are typically overridden when writing a game. Chapter 2 covered simple programs overriding just the setup method.
This section describes the *other* method that is typically overridden. What is the name of that method? What is that methods *signature*?
- (c) What value does the parameter have when FANG calls `public void advance(double secondsSinceLastCall)`? What is the *meaning* of the floating point number which is passed?
- (d) Approximately how frequently is the advance method called by the game engine?
- (e) If a very complex advance method took 0.1 seconds to execute could the game engine call the advance method at the normal rate?

3.2 Computer Program (Game) Design

It is possible to wander far and wide when introducing a section on software (or game) design. One of the first tasks is to track down a usable definition for design and the reader will recall how long it took to do that for *game* back in Chapter 1; *design* is almost as ubiquitous as game and it is used “technically” by many different disciplines (some fun on the different meanings can be found in Spillers's blog entry [Spi07]; “Yes, all 10 definitions!”).

Computer program design is the creation of a program to solve a particular problem. That is, it is the task *not* of solving the problem but rather of describing *how* the problem is solved. This is similar to being the author of a cookbook: you do not take a handful of fresh ingredients and create a fine meal; you describe how any competent cook could take a handful of fresh ingredients and create a fine meal.

What makes this design? What does it have in common with, for example, game design? It seems, looking at the various design fields, that they all have the designer indirectly creating experiences. That is, a graphic designer uses art and technique to create a poster (or computer interface or billboard or...) but the poster's purpose is to make the viewer want to purchase a particular brand of soft drink. Similarly, the computer programmer writes code that helps the user accomplish some task, solve some problem. There is a level of indirection in the thing being built.

So, what is a *game* designer building and what purpose does it have. Falling back on Jesse Schell's *The Art of Game Design* [Sch08], he claims that game designers build games in order to induce players to have a particular experience. He goes on to talk about the four basic elements of a game: aesthetics, story, mechanics, and

²Not all languages use the same syntax for this but all object-oriented languages have some form of inheritance, a way to base one class on another, already defined class.

technology. I mention these elements here because if we become good at software design these same elements can be applied (perhaps giving short-shrift to story).

Design is Problem Solving

Design is problem solving and, typically, solving problems at multiple levels simultaneously. You are describing a solution to a particular problem so you must understand its solution. You are expressing the solution in Java, a problem of its own. Juggling the solutions to multiple problems at multiple levels is difficult as simultaneous solution is complex.

How can we tame complexity? The answer is fundamental to all of computer science: work at multiple independent levels. That is, work at the very bottom level, defining the next more abstract level of commands and components that you need to solve your problem. Know what happens at the top level but put the details out of your mind. When you finish defining the next level up, go back and start over but with the next level up. This is known as *bottom-up* design.

Alternatively, start at the top level and write your solution *pretending that you have any necessary support methods*. That is, write the solution in terms of methods you have not yet designed. The important point here is that any method required to solve *part* of the top-level problem is, by definition, simpler than the original top-level problem. After you have the solution written in terms of another level of capabilities, design each of them in a similar manner. This is known as *top-down* design or *step-wise refinement*. This is the approach that we will use throughout the book.

Before going on, it is worthwhile to note that very few software designers in the real world use top-down or bottom-up approaches exclusively. With practice, it is possible to alternate between the two, thinking about the lowest level commands that will help solve the problem as well as decomposing the highest level problems. The two approaches often meet in the middle. The one important thing to keep in mind, however, is that the point of using different levels of abstraction is to permit upper layers to be implemented without knowing how lower layers are actually implemented. We will return to this topic in Chapter 6.

Problem Solving Techniques

At any level of solution, the facilities provided by the next layer down can be combined in five basic ways (as introduced in Section 1.5): sequence, selection, iteration, delegation, and abstraction. The next two sections will expand the first two of these techniques and the others will be presented in following chapters. The one important thing to note is that *delegation*, the use of named units of computation, fits right in with the idea of step-wise refinement. Looking at a particular level of a problem, we pretend that we have the building blocks to solve the problem. Each of those building blocks becomes, in turn, a named piece of computation, a *method*, which is called from this level down to the next.

These same techniques are used in game design: consider the rules of your favorite board game. Most turn-based games have a structure that looks something like this:

```
setup game
while (not game over)
    turn = next player
    player make move(s)
```

That looks familiar. Players making their move, that involves looking at the game (which constantly displays the public state of the game) and their “hand” (their private state), selecting a move (providing input) and resolving the move (updating game state: new position, successful or unsuccessful attack, etc.). This is, fundamentally, the same loop at the heart of every video game.

Notice that when you design a game, any game, you start with the general game you want to build, considering the four essential elements of aesthetics, story, mechanics, and technology, and then you describe how things work at that high level. Odds are, when considering the highest level of the game, you are not thinking about what color the dice should be or whether the attacker rolls first or second. Those decisions are further down and would be part of defining how a player makes their move.

Thus a hierarchical approach where details are pushed off to another level is used when designing games as well as when designing software.

Review

- (a) Designing a program and writing a program both involve problem solving, but at different levels. Which one is problem solving at a higher level (using instructions will less detail)?
- (b) What is the difference between top-down and bottom-up design?

3.3 Sequence

We have experience with sequence and even how it applies in Java. Look at the body of the setup method in any program in Chapter 2. Each contains blocks where a local variable is declared to name a new component and the component is created. Then, after creation, the location is set. Then, typically, the color is set. Then, finally, the new component is added to the game.

Most FANG games create sprites in this way: call the constructor, set all necessary attributes of the sprite, add sprite to the game. Many games create all of their sprites before the game begins. Thus this pattern is one we will see regularly.

The only difference between the setup methods of games in the previous chapter and `NewtonApple` is that `NewtonApple` does not declare names for the components inside of setup.

```

1 import fang.core.Game;
2 import fang.sprites.OvalSprite;
3 import fang.sprites.RectangleSprite;
4
5 /** NewtonApple pre-prototype */
6 public class NewtonAppleFixedAppleLocation
7     extends Game {
8     // An OvalSprite field to name the apple
9     private OvalSprite apple;
10    // A RectangleSprite field to name newton
11    private RectangleSprite newton;
12
13    // First pass at defining the sprites for the game
14    @Override
15    public void setup() {
16        apple = new OvalSprite(0.10, 0.10);
17        apple.setColor(getColor("red"));
18        apple.setLocation(0.50, 0.00);
19
20        newton = new RectangleSprite(0.10, 0.10);
21        newton.setColor(getColor("green"));
22        newton.setLocation(0.50, 0.90);
23
24        addSprite(newton);
25        addSprite(apple);
26    }
27 }

```

Listing 3.1: `NewtonAppleFixedAppleLocation.java`: a first pass

Lines 1 through 7 should be familiar from Chapter 2: the first three `import` three types from FANG, line 5 is a comment, and 6 and 7 declare the name of our `class` (which must match the name of the file in which it is defined) and what class this class extends. The end of line 7 is an opening curly brace, marking the beginning of the container which is the body of our class definition. The matching closing curly brace is on line 27.

Note that the lines between 8 through 26 are *indented*. This is not required by Java but is done to aid the programmer in seeing the containment relationship between the class and the rules within it. It is normal to indent each set of contained statements by one “indent level”. In the code listings of this book each level is 2 spaces; you may want to use a slightly greater indent if the formatting makes the code structure more obvious to you.

Lines 9 and 11 are new. They look something like the declaration of a local variable in one of the programs in the last chapter (see Listing 2.8, lines 17, 22, 27, and 32, for example) but not quite the same. Rather than being a name that only exists inside of `setup`, each of these is an attribute or a *field* of objects of this class. A field exists as long as the object of which it is a component exists; that is, `apple` and `newton` are visible inside of *all* methods declared in this class.

Both are **private** because only `NewtonApple`³ is permitted to see or change it. Yes, **private** and **public** (see line 15) are related; more on access rights below. The rule of thumb is: all *attributes* are declared to be **private**.

These lines, just like the **import** lines above, each end with a semicolon. Statements in Java end with semicolons (unless they end with a curly brace). This is one of the important parts of the artificiality of Java: the semicolons make it much easier for the compiler to take a program apart into its constituent rules for translation to bytecode.

Notice that an `OvalSprite` (and a `RectangleSprite`) is here referred to as an attribute of a game; a sprite is usually thought of as a component. This is an example of multiple levels: the oval *is* a component when viewed without context; in the context of the game, the component is one attribute of many composing the game.

Just as each component in a chess set has a type (the board, a black knight, a white queen, etc.), every component in Java must have a type. The type of the component is the name of a type or **class**, either one provided by Java or FANG or one written by you, the programmer.

The Java template for declaring an field is

```
<fieldDeclaration> := private <TypeName> <fieldName>;
```

This is a simplified template; Chapter 4 explains more about declaring components. Field declarations appear within a class but outside of all methods. The field name is visible *everywhere* within the container of the class, including inside of any and all methods.

Lines 5, 8, and 13 are special: they are *comments*. This book typesets comments in *italics*; many Java editors also display comments in a particular font/color combination to make them stand out. Remember that comments are treated as empty space by the compiler.

Comments are treated as blank by the compiler but they are very helpful for programmers reading code. This book will come back to commenting over and over. Remember when commenting to be *consistent* and to document (comment on) your *intent* in writing the given code.

Line 9 declares `apple`, a label for an `OvalSprite`. In previous programs, we combined, on the same line as the local declaration, the assignment of a newly created sprite to the name. Here we separate them so as to put all initialization of fields into the `setup` method.

The declaration creates a name which can be used to label a sprite; no sprite has actually been created. That is delayed until line 16 is executed and `new` is called to create a `OvalSprite`. The constructor for `OvalSprite` (specified with the name of the class and the parentheses with a parameter list inside) take the width and height of the sprite; we pass in the same value for both dimensions so result is a circle.

The difference between a component *type* or **class** and a component object is the difference between the blueprints for a house and a house constructed from the blueprints. If the blueprints are for a house in a subdivision, it might be the case that hundreds or thousands of specific houses are built from the exact same blueprint. Creation of the blueprint (like definition of the **class**) does not indicate that any houses (objects) of that type will be built. Similarly, purchasing a lot (defining a variable as in lines 9 and 11) for a given kind of house does not build the house. Instead, having a lot (variable) and having a blueprint (class) means that we can contract to have a house built (call to `new`).

³The multiple versions of `NewtonApple` defined in this chapter represent snapshots over time of the creation of the final game. While the complete name of the program will be used in captions of the listings, the name `NewtonApple` will be used to refer to the current incarnation generally when no confusion will result.

Line 15 declares a new rule, or *method*, for `NewtonApple`. The first line of a method declaration (where the method is defined) is called the *signature*. As we first saw last chapter, the signature has four parts:

- visibility : **public**
This means that components of types other than `NewtonApple` (the type we are currently defining) can “call” the method. There will be more on visibility in following chapters; for now, all methods should have **public** visibility.
- return type : **void**
Rules can calculate and return values. This rule returns nothing so we use the special component type **void** to indicate that no value of any kind is returned. This is enforced by the compiler.
- name : `setup`
The name used to call this method. Naming in a computer program is important enough to fill most of the next chapter.
- parameter list : `()`
Inside the parentheses appears a list of component types and names for values that this rule works with. This particular rule has an empty list.

The body of a new rule, just like the body of a new component, is contained inside curly braces (line 15’s `final {` matches with the `}` on line 26).

The Java method template is

```
<methodDeclaration> := public <ReturnType> <methodName>(<parameterList>) {
                        <methodBody>
                        }
```

A Different Method

How would we declare a helper method to *randomly* position the apple somewhere at the top of the screen to begin its fall? To begin writing a method, we need to know its signature; the signature of a method depends on its name, which, in turn, depends on *what* the method does.

The name and the purpose are intertwined. This is an example of bottom-up design because we need to be able to drop the apple at a different location each time the apple is dropped; the user has no meaningful choices if the apple always appears right above Newton.

So, we are going to have to randomly position the apple *each* time a new apple is dropped. So, we could call our new command `dropApple`. It does two things: positions the apple along the top edge of the screen *and* increments the number of apples that have been dropped. The second action requires a numeric value to increment. The number of apples dropped is used in creating the onscreen score display so it must be available in at least two different methods (whatever updates the score and `dropApple`). Thus we will need to define it as a field. The following snippet includes both the declaration of the field and the declaration of `dropApple`; identify the four parts of a method declaration in the listing.

```
29
30  /**
31   * on-screen text displaying the current score; contents are updated
84   applesDropped = applesDropped + 1; // another apple dropped
85  }
86 }
```

Listing 3.2: `NewtonAppleJustSetup.java`: `dropApple` method

Notice the addition of useful comments; they are sometimes distracting in listings in the book but it is always a good idea to explain what the code you are writing is doing. Remember to document *intent*: what does the code do and why does it do it.

Notice the *type* of the `applesDropped` field (declared on line 38). It is of type `int`. This is the built-in Java numeric type representing *integers*. Integers are whole numbers such as 1, -3, 0, and 1030369; notice that there are no markers for the millions or thousands in the last number. Also notice that nowhere in these examples is there a decimal point.

The other numeric type we will use in this book is `double`. Double numbers represent values with a decimal point in them and a possibly non-zero fractional part. Examples include 1.4, -90.44, 104.00, and 1030369.9991; the third number has a zero fractional part but it is *not* an integer. The next chapter will go into greater detail on different numeric types.

The `dropApple` method is called from `setup` (see Listing 3.3) after `apple` and `applesDropped` are initialized. Line 93 sets the location of the apple sprite (using `setLocation` as we saw in the previous chapter). The second parameter, the y-coordinate for the apple is clear enough: 0.00 is the top of the screen (as in line 18 of `NewtonsAppleFixedAppleLocation.java`, Listing 3.1). What is `randomDouble()`?

The `Game` class provides more than just color creation methods. It also provides input methods (as we will see below when we sample the mouse's location), networked input methods (so we can get input from multiple players), and even random number generation. What is *random number generation*? Think about flipping a coin or rolling a die. A single die provides an integer on the range [1-6] (the square brackets mean the range is inclusive). Most programming languages provide a way to get a random number on such a range. FANG's `Game` class puts a wrapper around the standard Java way (which we will get to by the end of the book) by providing `randomInt` and `randomDouble` methods.

Looking at `randomDouble`, if it is called with no parameters, a random fraction on the range [0.0-1.0) (the round bracket means 1.0 is *not* included). Called with one `double` parameter, `d`, `randomDouble` returns a `double` on the range [0.0-d) (a non-negative number less than `d`) and finally, given two `double` parameters, `m` and `n` (such that `m` is no larger than `n`), `getDouble` returns a value on the range [m-n).

The following sample lines have the range of numbers returned in comments:

```
randomDouble()           // [0.0-1.0)
randomDouble(100.0)     // [0.0-100.0)
randomDouble(-1.0, 1.0) // [-1.0-1.0)
```

So, what is `randomDouble()` doing in line 93? It is providing the x-coordinate for the apple. The method, called with no parameters, returns a number on the range [0.0-1.0), exactly the range of the x-coordinate *on the screen*. Thus the apple will be positioned at some x-coordinate at the top of the screen. The method `dropApple` can be used whenever we need to start a new apple falling: the value returned by `randomDouble` is selected anew each time it is called.

Finishing setup

Setting up our game is complicated. The declaration of all of the fields and their initialization in `setup` is longer than any program we have yet examined. The listing below shows the declaration of all of the fields and the complete `setup` method for `NewtonsApple`:

```
16  /** on screen representation of the apple; a small red circle */
17  private OvalSprite apple;
18
19  /** on screen representation of Newton; a small green square */
20  private RectangleSprite newton;
21
22  // ----- keeping and displaying the score -----
23
24  /** number of apples caught */
25  private int applesCaught;
```

```

26
27 /** number of apples dropped */
28 private int applesDropped;
29
30 /**
31  * on-screen text displaying the current score; contents are updated
32  * whenever the score changes
33  */
34 private StringSprite displayScore;
35
36 /**
37  * The method called by FANG before the game starts. Include all
38  * "one-time" instructions. Our simple computer games will create and
39  * add Sprites here the game begins. Create and add Sprites; Sprites
40  * are visible things in the game. Once they are added to the game
41  * they are automatically redrawn by the game engine.
42  */
43 @Override
44 public void setup() {
45     // initialize the score
46     applesCaught = 0;
47     applesDropped = 0;
48
49     // The apple is small and red; its initial position is set randomly
50 // in the dropApple routine (called here and when apple bottoms out)
51 // our apple is a circle (x and y dimensions are the same)
52     apple = new OvalSprite(0.10, 0.10);
53     apple.setColor(getColor("red"));
54     dropApple();
55
56     // newton is small, green, and begins at the middle bottom of the
57 // screen.
58     newton = new RectangleSprite(0.10, 0.10);
59     newton.setColor(getColor("green"));
60     newton.setLocation(0.50, 0.90);
61
62     // The score is in the upper left corner of the screen in white
63     displayScore = new StringSprite(0.10);
64     displayScore.setColor(getColor("white"));
65     displayScore.topJustify(); // move location to upper-left corner
66     displayScore.leftJustify();
67     displayScore.setLocation(0.00, 0.00);
68     displayScore.setText("Score: " + applesCaught + "/" +
69         applesDropped);
70
71     // for the sprites to be part of the game, they must be added to the
72 // game itself.
73     addSprite(apple);
74     addSprite(newton);
75     addSprite(displayScore);
76 }
77
78 /**

```

Listing 3.3: NewtonsAppleJustSetup. java: setup method

Notice that there are five fields declared: `apple`, `newton`, `applesCaught`, `applesDropped`, and `displayScore`. The three sprites exactly match those we described in the design at the beginning of the chapter.

The two integer fields keep score for our game: `applesCaught` will count the number of apples the player has caught and `applesDropped` will count the number of apples dropped. We will initialize then both to 0 and then, whenever an apple is dropped, we will increment the number dropped and whenever an apple is caught we will increment the number caught.

The setup method has in-line comments explaining what is going on. The first set of statements, lines 55-57, set both parts of the score to 0: no apples have been caught and no apples have been dropped. It is important to make sure that values are assigned to *all* variables (fields and local variables) before they are used.

Lines 54-62 are the same as setup lines in Listing 3.1: `apple` and `newton` are constructed, colored, and set to their initial locations.

Lines 65-70 construct the score sprite (a `StringSprite`), set its color, text, and location. Lines 67-68 move the location of the `StringSprite` to the upper-left corner. Only `StringSprite` supports these methods (along with `justifyCenter` to return the location to the center point as well as right and bottom versions). This makes it easier to match up strings and keep them in a given location even when their text value changes (and might change width). By putting the location in the upper-left corner and anchoring that in the upper-left corner of the game screen (look at the location in line 69), no matter how many digits the score grows to, the word “Score:” will stay in the same place.

Finally, setup ends by adding all three sprites to the game. Remember the stacking order; if the apple is placed overlapping with the score, which one will be in front? How would you reverse that?

Movement

With all the sprites in the game, how to we make something move? It is finally time to override `advance`.

```
93 }
94 }
```

Listing 3.4: NewtonsAppleFallOnce. java: advance method

The parameter passed into `advance` is the number of seconds since the last call. It is a fraction of a second. Since the computer may have other tasks to accomplish, there is no assurance that the main video game loop takes exactly the same time to execute every time. To avoid being bound by the speed of the computer, FANG provides the actual elapsed time to this method.

Line 101 calculates how far the ball should have moved in the small amount of time between frames. In physics, $\text{displacement} = \text{velocity} \times \text{time}$. Velocity is distance per time so, in our case, we set it to a tenth of a screen per second (or ten seconds to fall one screen). This is too slow to make for a challenging game but it permits you to see the ball falling.

Compile and run `NewtonsAppleFallOnce. java`. When you run it, you will see:

Note that the horizontal position of the apple (the circle half visible along the top of the screen) is *randomly* selected so it is unlikely that yours will be right where it is in the picture. Note, in your running program, that the apple does not move. We overrode the `advance` method with a line to move the apple every frame but it is not moving. What is wrong?

Look at the lower right of the screenshot: the button there says **Start**. FANG runs setup and then waits for the player to press **Start** before starting the game loop. When you press the button, you will see the apple fall, slowly, down the screen:

While a static image in a book does not really capture what happens when the apple falls down the screen, Figure 3.3 shows what happens when the game has been started. Note the button in the lower left now says **Pause** (FANG lets you pause a running game) and the ball is further down the screen⁴. The apple will take

⁴If you compare Figure 3.2 to Figure 3.3 you will notice that the apple does not line up vertically. The images come from two different runs of the program and the apple was randomly placed at the beginning of each run.



Figure 3.2: NewtonsAppleFallOnce game *before* starting

about 10 seconds to fall off of the screen and it will then just keep falling. We need to finish designing and writing the advance method and to do that we need to be able select different courses of action.

Review

- (a) Why do programmers indent their code blocks?
- (b) In which methods can **private** fields be used?
- (c) Is `OvalSprite` a class or an object?
- (d) Is `SomethingYouHaveNeverHear dOf` a class or an object?
- (e) What does **void** mean?
- (f) Name two numeric types (these are types capable of storing a number).
- (g) `PolygonSprite` is one `Sprite`-extending type. Name three other `Sprite`-extending types (these are types capable of holding 2D graphic and geometric information).
- (h) What does the `StringSprite` method `topJustify` do?
- (i) What does the `StringSprite` method `leftJustify` do?



Figure 3.3: NewtonsAppleFallOnce game *after starting*

(j) Compile and run NewtonsAppleFallOnce. Now change line 94 to `apple.translateY(0.10)`; What happens? Now move this line to be the last line of the setup method (advance should now have nothing between the braces). What happens now? Explain the difference.

3.4 Selection

To complete our game, the advance method must do four things:

1. Move Newton to where the player's mouse is
2. Move the apple down the screen
3. Check if apple is caught
4. Check if apple hit ground

The code to do this is a little longer than what we have written so far. To make it clearer, we will use these steps as comments to structure our code:

```

96     if (position != null) {
105         applesCaught = applesCaught + 1; // another apple caught
113         displayScore.setText("Score: " + applesCaught + "/" +

```

Listing 3.5: NewtonsApple: structure of advance

(Notice the line numbers; the method will be filled in around these comments.)

No Such Object

Before filling in the code in the above template, we will take a slight detour. Consider a modified version of checkers where we want to reward the player who last captured an opposing checker. At the beginning of each player's turn, if they were the last to capture a checker, they get a cookie. If we have a special square beside the board where we put the last captured checker, then looking at it at the beginning of each turn, the moderator can determine whether or not the current player gets a cookie.

What "value" does the "last captured checker" have at the beginning of the game? In fairness, which player should be rewarded with cookies before either has captured a checker? We need a special value that indicates that there is no such component, no "last captured checker."

In Java, the special value meaning no such component is `null`. This is different than a rule that returns `void`: `void` means that no value is ever returned; a rule that returns a component can return `null` to indicate that there is no such component (right now; if you call it again the return value might change). Back from the detour.

To move newton, we need to get the player's mouse location. Then, extract the x-coordinate from the mouse location. Then, set the x-coordinate of the sprite to the new value. This is trickier than it sounds: What if the player stared the game but has moved their mouse out of the game to answer an IM? Then there is no location for the mouse. This is signalled by returning `null` as the mouse's current location. We will leave newton alone if the mouse has no location.

```

96     if (position != null) {
97         newton.setX(position.getX());
98     }
99
100     // (2) Translate apple down screen; velocity * time = distance

```

Listing 3.6: NewtonsApple: Moving Newton

The `!=` symbol reads as "not equal to" so, if `position` is not `null`, set `newton`'s x-coordinate to the `position`'s x-coordinate. The `if` statement is how selection is written in Java.

The if Statement

Selection is choosing one rule over another. In checkers, **if** you jump over an opposing checker, **then** you get to remove the checker. Or, in chess, **if** the king is under threat and there is no way to get him out of threat, **then** the player whose king is under threat loses.

Each of these two examples have the rule follower either doing or not doing something. It is also possible to select from two (or even more) different paths: **if** the temperature is above sixty degrees Fahrenheit, **then** where shorts or **else** where long pants.

Where does this come into play in `NewtonsApple`? Three places: we must make sure the user's mouse location exists before we can use the dot notation with it; we need to be able to check if Newton caught the apple and adjust the score and drop a new apple if so; we need to check if Newton missed the apple, it hit the ground, and adjust the score and drop a new apple if so. We will examine the `if` statement shown in the previous section, present the Java template for `if` statements, and then explore Boolean expressions.

```

98     }
99
100     // (2) Translate apple down screen; velocity * time = distance

```

Listing 3.7: NewtonsApple: First if statement

Lines 98-100 of `NewtonsApple` are the `if` statement that checks whether the player has a mouse position. As was described above, it is possible for the mouse to be in some other application's window. FANG signals the lack of a usable position by returning the special `null` value. One thing you should *never* do with a `null` value is use the dot notation with it. That is, if `position` were `null`, then `position.x` would cause Java to halt the program with an exception.

If execution were simply sequential, then having line 107 appear in advance would mean that it *must* execute every frame (a method executes from top to bottom when called *unless* the flow of control through the method is modified by selection or iteration statements). The `if` statement on line 116 evaluates a *Boolean expression*, an expression which is either `true` or `false`, and then executes the body of the `if` statement if and only if the expression is `true` when evaluated. Whether the body is executed or not, when the `if` statement finishes, execution continues with the next executable line in the method.

The Java template for the `if` statement is

```
<ifStatement> := if (<booleanExpression>
                <thenStatement>
```

The `<thenStatement>` is either a single statement (which then ends with a semicolon) or it is a block of statements enclosed in curly braces. To keep our `if` statements clear, we will use curly braces even for single statements in this book.

There is an alternate template for the `if` statement, the `if/else` version of the statement:

```
<ifStatement> := if (<booleanExpression>
                    <thenStatement>
                    else
                    <elseStatement>
```

Both statements here are single statements or blocks in curly braces. The meaning of the `if/else` version is that Java evaluates the expression, and if it is `true`, the `<thenStatement>` is executed and the `<elseStatement>` is skipped over; if the Boolean expression is `false` the `<thenStatement>` is skipped over and the `<elseStatement>` is executed. This lets Java select from one or another rule depending on some Boolean expression.

Boolean Expressions

What is a Boolean expression? It is an expression with a truth value. One form of Boolean expression is a comparison of two values. Consider the following snippet of code:

```
int x = 13;
if (10 < x) {
    x = x - 1; // Line A
}
```

Does Line A get executed? Yes, because the `<` operator means “less than” and it applies to numbers just as you would think that it would: it returns `true` if the left hand expression is less than the right hand expression and `false` otherwise.

Java has six standard comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. These read “is equal to”, “is not equal to”, “is less than”, “is less than or equal to”, “is greater than”, and “is greater than or equal to”, respectively.

Two character operators must be typed with no space between the two characters. That is, less than or equal to is typed `<=`; the sequence `<~=` is interpreted as the operator less than followed by the assignment operator (and, since that sequence of operators makes no sense, the Java compiler will complain).

Pay special attention to the “is equal to” operator, `==` and do not confuse it with the “is set equal to” or assignment operator, `=`. The first compares two expressions, returning a truth value; the second assigns the name on the left of the operator to be a name of the expression result on the right. Thus `x = 13` in the above snippet assigns 13 to the variable `x` while `x == 13` would return `true` after the above assignment. Java will flag the use of `=` in a Boolean expression as an error but you should watch to make sure you type what you mean.

Another type of Boolean expression is a call to a method which returns a Boolean value. For example, the `Game` class has a method called `isGameOver()`. This method takes no parameters and returns `true` if the game is over and `false` otherwise. The main video game loop uses this Boolean method to determine whether to do the loop again or not. Actually, it uses the logical inverse of this method. To invert a Boolean expression, making `true` `false` and *vice versa*, place a `!` in front of the expression (and read it as “not”). Thus to have an `if` statement based on the game continuing, you could have

```

if (!isGameOver()) {
    someStringSprite.setText("Game Continues");
} else {
    someStringSprite.setText("Game is OVER");
}

```

Knowing these two ways of getting Boolean values, we know enough to finish expanding the four requirements of advance into actual code.

Review

(a) The statements of a method body are executed in sequential order unless specified otherwise. Suppose you sometimes want a statement to execute and other times do not want it to execute. How could you indicate that the order of statement execution should not be sequential and instead should execute the statement depending upon a condition.

(b) When evaluating a Boolean expression in Java, what are the two possible outcomes?

(c) What is the difference between `score = 2` and `score == 2`?

3.5 Finishing NewtonsApple

We have examined the code for moving Newton to the same x-coordinate as the player's mouse; we even decided that we didn't need to do anything if the mouse was not available to our game.

We saw how to move the apple; now we need to decide on its velocity. We want the game to be easy so we will use a fall rate of one screen every three seconds or 0.33 screens/second⁵. The amount that the apple falls in any given time period is still `velocity * time`.

```

102
103    // (3) Check if newton has "caught" the apple

```

Listing 3.8: NewtonsApple: Moving apple

Sprites can detect intersection with other sprites. This is done using a Boolean method called `intersects` which takes as a parameter another sprite to test for intersection with. This method lets us check if Newton caught the apple. If the apple is caught, we will update the number of apples caught and drop a new apple.

```

105    applesCaught = applesCaught + 1; // another apple caught
106    displayScore.setText("Score: " + applesCaught + "/" +
107        applesDropped);
108    dropApple();
109    }
110
111    // (4) Check if the apple has hit the ground (y-coordinate >= 1.0)

```

Listing 3.9: NewtonsApple: Catching apple

We know the apple was missed if the apple's location reaches the ground. Or, when the y-coordinate of apple is greater than or equal to 1.0 (the bottom of the screen).

```

113    displayScore.setText("Score: " + applesCaught + "/" +
114        applesDropped);

```

⁵Constant velocity is *not* how things fall in the real world; it seems that this simplification might keep Newton from discovering gravity!

```
115     dropApple();
116   }
117 }
118 }
```

Listing 3.10: NewtonsApple: Missing apple

That completes advance by updating the state of the game: the two movable sprites are moved (one only if there is a viable new location) and the two scoring conditions are checked. The score is updated if necessary and a new apple is dropped, again, if necessary.

A note on “dropping a new apple”. That phrase might be misleading. This game does something that a lot of games (and other computer programs for that matter) do: it reuses an already created resource when that resource is no longer in use by the user. That is, there is really only one apple in the game. When it is caught or goes splat, that exact same apple is teleported to the top of the screen and magically becomes the *new* apple. The player does not care about this (the old apple is no longer of interest) and we are not required to call `new` all of the time.

Review

(a) What `Sprite` method is used to determine if two sprites overlap each other?

3.6 Summary

For a player’s choices to be meaningful, a game must have different outcomes depending on the sequence of choices the player makes. Not every choice leads to a different ending but some sequence of choices must. This requires the rules of the game to include selection as well as sequence.

Computer Program Design

Design is problem solving. Designing a computer program is not quite solving the potential user’s problem but instead describing *how* to solve that problem to a computer. Then the user can run the program and solve as many instances of the given problem as they like.

Game design, like program design, is an indirect approach to creating something: the game designer is designing an experience, the experience the user has while playing. Yet the designer can only specify the game the user will play. Games can be thought of as having four basic elements: aesthetics, story, mechanics, and technology.

Problem Solving Techniques

Simple problem solving techniques can be combined together using five different methods:

- *sequence* do things in order they are written
- *selection* choose, based on some criteria, which rules to follow
- *iteration* do some set of rules over and over again
- *delegation* collect some set of rules into a named subprogram
- *abstraction* encapsulate rules and components together into a new type

Java Method Signatures

In Java, a named subprogram is called a *method*. A method is declared with a *signature* and a *body*, the body containing the set of rules run when the method is called.

The four parts of a method signature is

- *access level* so far methods are declared **public**
- *return type* the type returned by the method or **void** if there is no type to return
- *name* the name of the method
- *parameter list* in parentheses, the collection of type/name pairs representing

Selection

In Java, selection is done with the **if** statement. The **if** statement takes a *Boolean* expression, an expression which is either **true** or **false**. If the expression is true, then the body of the **if** statement is executed. If the expression is false, then the body of the **if** statement is skipped; if there is an **else** clause, the body of the **else** is executed.

FANG Classes and Methods

Java Templates

```
<fieldDeclaration> := private <TypeName> <fieldName>;
```

```
<methodDeclaration> := public <ReturnType> <methodName>(<parameterList>) {
    <methodBody>
}
```

```
<ifStatement> := if (<booleanExpression>)
    <thenStatement>
```

```
<ifStatement> := if (<booleanExpression>)
    <thenStatement>
    else
    <elseStatement>
```

Chapter Review Exercises

Review Exercise 3.1 What type only has values **true** and **false**?

Review Exercise 3.2 Describe the *video game loop*.

- What are the three parts of the video game loop?
- What *methods* does FANG permit you to override to hook into the video game loop?

Review Exercise 3.3 What is a *Boolean expression*?

Review Exercise 3.4 What is a *block*?

Review Exercise 3.5 Start with `NewtonsApple.java`. Modify the program to drop your first name rather than a red `OvalSprite`. The gameplay remains unchanged.

Review Exercise 3.6 The `if` statement is Java's version of which of the problem solving techniques?

Review Exercise 3.7

- (a) How many times can the `setColor` line in the following code be executed? Give all possible values.

```
public void setup() {
    double x = randomDouble();
    RectangleSprite rs = new RectangleSprite(1.0, 1.0);
    if (x < 0.5) {
        rs.setColor(getColor("dark violet"));
    }
    addSprite(rs);
}
```

- (b) What are the possible colors of `rs` in the above listing?
 (c) What is the scope of the variable `rs`?

Review Exercise 3.8 What does `null` mean in Java?

Review Exercise 3.9 What can you *not* legally do with a `null` reference in Java?

Programming Problems

Programming Problem 3.1 It is mentioned in the chapter that `NewtonApple` uses an unrealistic model of gravity.

- (a) The given game uses constant velocity. What is a more realistic model of gravity?
 (b) To implement constant acceleration, you would need to replace the constant screen velocity (in advance) with a variable.
 ii. Declare a *field* in the class called `velocity` with the type `double`.
 ii. In `dropApple`, initialize the value of `velocity` to `0` (screens/second).
 ii. In advance, replace the use of the constant `0.33` with the field. (Question: What would the program do if you compiled and ran it now? If you're not sure, why not try it?)
 ii. Also in advance, increment `velocity` by `0.16`.

Programming Problem 3.2 Given that the `Game` class has a `randomColor()` method which returns a `Color` (the type of object expected by `Sprite.setColor` method)

- (a) *Where* would you change the code so that each time the apple was restarted at the top of the screen it was set to a random color?
 (b) Go ahead and add the required line to the right method.

Programming Problem 3.3 Start with `NewtonApple.java`.

- (a) Modify the program replacing `newton` with a heart (see Exercises in Chapter 2 for a simple heart design). Make sure that the whole sprite moves with the mouse.
 (b) Make the apple "fall" from the right side of the screen to the left side of the screen.
 (c) Make the heart move vertically on the left edge of the screen.
 (d) Replace the `OvalSprite` with an `ImageSprite` of your favorite animal.

This changes the meaning of the game, does it not? From discovering gravity to finding loving homes for cute little animals. This idea of *skinning* or *rebranding* a game with just a change in the graphics (and here, a rotation of the gameplay through 90 degrees), is often used for Web based "advergaming", games designed for advertising. Ian Bogost's *Persuasive Games* [Bog07] has more on this phenomenon along with some interesting insight into what happens to the game with a new skin.

Programming Problem 3.4 Start with `NewtonsApple.java`. Modify the program so that instead of limiting newton to moving horizontally, newton moves with the mouse all over the screen. This is, in many ways, easier than limiting the movement to a single dimension.

Programming Problem 3.5 Design a pursuit/evasion game similar to `NewtonsApple`. The “apple” should begin falling above the player’s square. Then, when the player moves out of line with the falling object, it should correct its trajectory by moving 0.05 screens toward the player. That is, if the player’s x-coordinate is smaller than that of the apple, the apple should subtract 0.05 from its x-coordinate this frame. The same if it is greater. The player starts with 5 lives and loses one each time the falling item touches them.

This assignment purposely did not describe what “game” is being played. Can you provide a credible *backstory* about the game to put the gameplay into an interesting context? Would the backstory be helped if you changed the same of either the item falling or the player’s representation? See Bogost’s *Persuasive Games* [Bog07] for more thoughts on meaningful play.

Components: Names, Types, Expressions

We have examined what a game *is*: a collection of components, each with some number of attributes, and a collection of rules determining how the components interact. The combination of the components and rules also provide the player with meaningful choices which impact the outcome of the game.

In Chapter 3 we wrote our first game, a Java program that made extensive use of the sequence and selection techniques of combining problem solutions. We also started to figure out how to design programs in a top-down manner. This chapter continues using the top-down design approach, creating a game with multiple cooperating classes; it also presents more information on declaring fields and local variables.

4.1 Randomness in a Game

Have you ever considered how many games have an element of randomness in them? If you spin a spinner, throw dice, flip a coin, or shuffle cards, the game is using randomness. It is often the case that games present the player with the “hand they were dealt” and the point of the game is to use skill to overcome any adversity handed out by the random resource.

In the last chapter we used randomness in a FANG game: the apple in `NewtonsApple` was randomly positioned across the width of the screen each time it was dropped as a *new* apple. The fun of the game was seeing if you can overcome the constantly shifting apple’s location and still catch it.

The game in this chapter is a simplified version of the dice game craps. Dice games, historically, are wagering games [Kni00]; for our purposes, the game will be played for a pile of matchsticks.

What do Dice Do?

What do dice *do*? Think about this from the point of view of what they provide to the game. A die is a component of a game which itself has state (the face that it is showing) and the ability to change its state (by rolling). We will use this simplified view of a die to go about designing one to play a dice game.

EasyDice Rules

EasyDice is a game with two six-sided (cube) dice. The player makes a wager on their “winning” the current turn and then the turn begins. The *value* of a throw of the dice is the sum of the pips on the face of the two dice. Thus rolling a five and a three would give the value of eight to the roll.

The player's initial roll of the dice either wins the turn immediately or sets the value rolled as the *point* for this turn. An initial value of seven or eleven wins a turn of *EasyDice* on the first roll. Any other value is the point for the remainder of the turn.¹

When a turn continues with a point, the player continues to roll until either a value of seven or a value of the point is rolled. The player wins if the next number rolled is the point and the player loses the turn if the value rolled is seven. Winning the turn means receiving double the wagered number of matchsticks back while losing means losing all the wagered matchsticks.

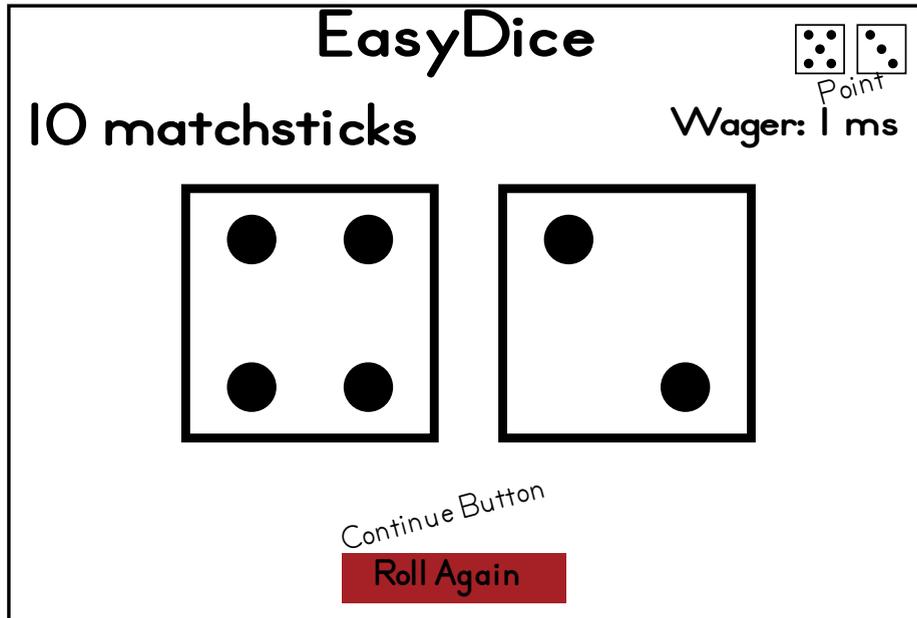


Figure 4.1: EasyDice game design diagram

In Figure 4.1 you can see a simple layout for the *EasyDice*: the current roll of the dice is in the middle, the point (from the first roll) is recorded on two dice in the upper-right corner of the screen, and the player continues to make rolls by pressing the “button” in the middle of the screen.

Note the quotes around “button”: that is really a rectangle and a string displayed one over the other on the screen². To press the button, the player clicks within the box. The game waits for a button to be pressed by checking for a mouse click within the `RectangleSprite` representing the button. When the button is pressed, the state of the game is changed.

Simpler Dice

These dice (both large and small) look dauntingly complex. Each is a `RectangleSprite` with a collection of `OvalSprite`s in a contrasting color displayed above them. Depending on the number being displayed, between one and six different `OvalSprite`s are displayed with coordinates and sizes depending on the coordinates and sizes of the dice they are part of (and the number being displayed). We will have to leave dice with pips for a later chapter; instead we will simplify them to show a large digit representing the number of the face showing. Thus Figure 4.2 shows the same game state as the previous figure with the face of each die was replaced with a digit representing its value.

The simplified version is developed by the end of this chapter; the version with pips and dice faces is an exercise in chapter 6.

¹This is the biggest simplification in *EasyDice*: in craps there are a number of values (two, three, and twelve) which will immediately end the turn with the player *losing* the turn. Additionally, there are a lot of different side bets possible in a real craps game.

²This description applies to buttons in most graphical user interfaces, too.



Figure 4.2: EasyDice game design with written out numbers

Cooperating Classes: Our Own Sprites

FANG, like almost any software framework, is not a game but rather is a game construction kit. It comes with some low-level pieces which can be put together to make game components. Looking at the description of EasyDice, it seems that we will need three different classes: the game class, EasyDice, the die class, OneDie, and a button class, EasyButton. With one button and only four dice, it would be possible to manipulate the sprites and values for each of these objects directly within the game class, EasyDice. That would confuse things at different levels, making the whole program must more difficult to implement.

What we will do is we will define the *public interface* for each of the two component classes. Then, using the public interface, we will write a demonstration program which rolls two dice over and over. The point of doing that here is that we don't have to know how the dice or button classes are implemented to *use* them. We will then work our way through implementing the classes through stepwise refinement.

Defining a Public Interface

The public interface of a class is the collection of all public methods and fields that it has. It is the set of all services which can be requested from that class including any public constructors.

The public interface is the *interface* between two different levels of abstraction. When we look down on OneDie from the point of view of EasyDice, we don't care what is inside the die or *how* it works. We just need to know what we can ask it to do.

Earlier we discussed what a die can do: it displays a number and it can generate a new number. In Java, we will also need to be able to construct one and set its color, location, rotation, and all of that.

Wait! If we *extend* a *Sprite* derived class then we will get much of the appearance methods for free. Thus we can consider the public interface to be:

```
public class OneDie
    extends ...Sprite {
    public OneDie(...) ...
    public int getFace() ...
    public void setFace(int newFace) ...
```

```

public void roll() ...
}

```

The constructor might take parameters; that remains undecided. The `getFace` method returns the current face which is up on the die and `roll` rolls the die, randomly changing the face which is on the top. Those three methods provide the *services* ascribed to a die above.

Considering `EasyDice` there is one service we missed in discussing a die: you can pick it up and put it down with a *given* face on top. You would do this when using a die to keep score or to indicate the point. This fourth service is provided through the **public** `setFace` method.

These four methods, all declared to be **public**, are the only visible methods in our `OneDie` class. Looking at the list, we have no idea how any of them work. But we don't care because we can use `OneDie` without knowing any more. Before we do that, however, we will try our hand at designing another public interface.

What is a Button?

The previous description of a button is that it is a rectangle with some text displayed above it. What is a button's public interface? A button can be constructed, it can do all the things sprites can do, it can change the text displayed on it, and it can tell you whether or not it was pressed. There is not much else that you would want a button to do. So, the public interface for `EasyButton` is:

```

public class EasyButton
  extends ...Sprite {
  public EasyButton(...) ...
  public void setText(String message) ...
  public boolean isPressed() ...
}

```

When we create a button and put it on the screen, we will call the `isPressed` method to see if it has been pressed. On any time through the game loop where the user pressed the mouse button over the screen button, that method will return **true**³.

Again, we don't need to know more than this to use an `EasyButton`.

Designing with a Public Interface: RollDice

To show that we don't need to know how dice or buttons work to use them, let's design a demonstration of our `OneDie` class. We will display two dice and a button. When the user presses the button, the game will roll the dice again. Not much of a game as the user's choice to press the button has no impact on the outcome (the numbers change) but it is a fine demonstration of the new classes.

```

1 import fang.core.Game;
2
3 /**
4  * Demonstration game using OneDie and EasyButton classes: Roll two big
5  * dice every time the button saying "Roll Dice" is pressed.
6  */
7 public class RollDice
8   extends Game {
9   /** the button at the bottom of the screen */
10  private EasyButton button;
11
12  /** the left die */
13  private OneDie leftDie;
14
15  /** the right die */

```

³The **boolean** type is the type used to return Boolean truth values.

```
16 private OneDie rightDie;
17
18 /**
19  * Advance the game one frame.
20  *
21  * @param secondsSinceLastCall time since last advance
22  */
23 @Override
24 public void advance(double secondsSinceLastCall) {
25     if (button.isPressed()) {
26         leftDie.roll();
27         rightDie.roll();
28     }
29 } // advance
30
31 /**
32  * Set up the game for play. Initializes all of the sprites (either
33  * here or in other setup functions).
34  */
35 @Override
36 public void setup() {
37     button = new EasyButton();
38     button.setScale(0.5);
39     button.setLocation(0.5, 0.85);
40     button.setColor(getColor("yellow"));
41     button.setTextColor(getColor("navy"));
42     addSprite(button);
43
44     button.setText("Roll Dice");
45
46     leftDie = new OneDie();
47     leftDie.setScale(0.33);
48     leftDie.setLocation(5.0 / 18.0, 0.5);
49     addSprite(leftDie);
50
51     rightDie = new OneDie();
52     rightDie.setScale(0.33);
53     rightDie.setLocation(13.0 / 18.0, 0.5);
54     addSprite(rightDie);
55 } // setup
56 } // RollDice
```

Listing 4.1: RollDice demonstration program

Lines 9-16 declare three attributes of our game. Two things to notice: the names of the classes (which we are assuming have the public interfaces described above) begin with capital letters (more on Java naming conventions below in Section 4.3) and, if you look back at the top of the program, these two classes have no **import** statements.

Java looks for named classes either in the classpath (as we saw in the examples in Chapter 2) or, if there is no **import** statement, it looks in the current directory. Thus if we define `OneDie` and `EasyButton` in the same directory as `RollDice`, we don't have to specify any import statements. This feature is made possible by the strict naming requirement that class names and file names match exactly.

Lines 36-55 are `setup`; `setup` runs before `advance` but sorts after `advance` in sorted source code. The constructors for our new classes, as used here, take no parameters. The rest of the lines here make sense, setting

the color, the location, and the scale of the various components on the screen. Only line 41 is odd; what does `setTextColor` mean in terms of an `EasyButton`?

A button is composed of two different visual elements: the rectangle and the text. When designing a composite type, if you are going to let programmers set the colors of some of the constituent parts, you want to let them set the colors of all of them. Thus `setColor` will set the color of the rectangle and `setTextColor` will set the text color. It is often the case that the first time you go to use a newly designed public interface you find that there is no way to do something you want to do. It is good to notice this *before* implementing the entire new class.

Finally, lines 24-29 are `advance`. All `advance` does is check if the button has been pressed. If it has, the two dice each roll themselves. This method is short and sweet. That this is so easy to write gives confidence in the public interfaces we have defined.

Now, how do we create an `EasyButton`? The problem is that we need some way to compose two sprites together into a single sprite.

4.2 One More Sprite: `CompositeSprite`

FANG permits the *composition* of multiple sprites together into a single sprite. If you have experience working with vector drawing programs and have ever used the **Group Objects** command, you have some idea what this is like.

A `CompositeSprite` is a sprite that behaves like a `Game` in that it supports `addSprite`. You can add as many sprites as you want to a `CompositeSprite` and then, with a reference to the whole composite, you can move, rotate, or scale the whole group of sprites at the same time.

A Screen within a Screen

A `CompositeSprite` is almost identical to the screen you see. Other sprites can be added to it. The center of the composite (and thus the location of the composite) is $(0.0, 0.0)$ *inside* the composite sprite. You set the location of objects in the composite relative to this point. You can then set the location of the whole composite relative to the game screen.

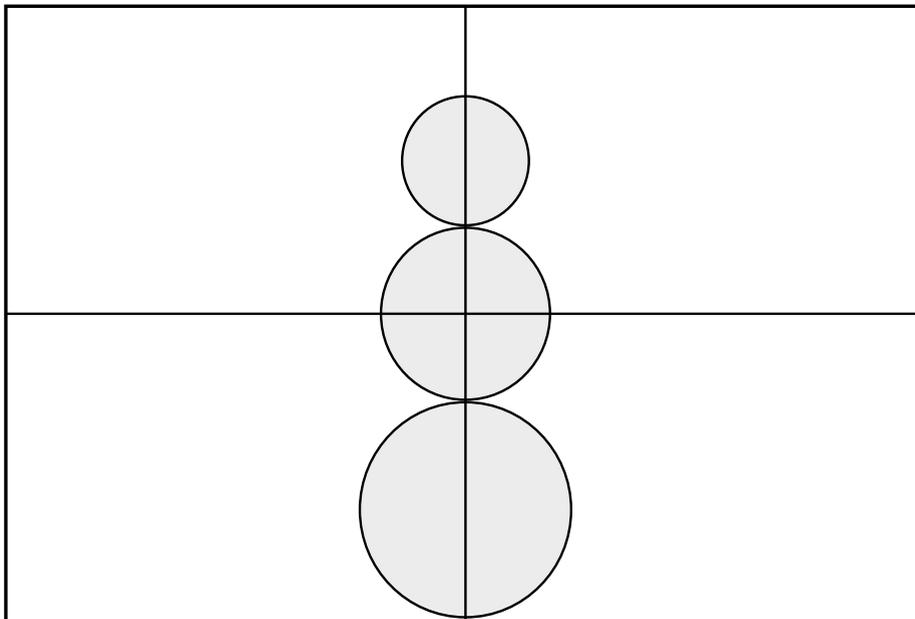


Figure 4.3: Snowman `CompositeSprite`

Figure 4.3 shows the design of a snowman CompositeSprite. The x-axis and the y-axis are shown in the drawing along with three OvalSprites. The important thing to note is that the positions of the three “snowballs” are along the y-axis, spaced out so they are just touching.

```

1 import fang.core.Game;
2 import fang.sprites.CompositeSprite;
3 import fang.sprites.OvalSprite;
4
5 // FANG Demonstration program: CompositeSprite
6 public class Snowman
7     extends Game {
8     private CompositeSprite blueSnowman;
9
10    // spin the snowman about 4 times a minute; see how everything
11    // moves relative to (0.0, 0.0) ON the snowman; 24 degrees/second
12    @Override
13    public void advance(double secondsSinceLastCall) {
14        blueSnowman.rotateDegrees(24.0 * secondsSinceLastCall);
15    }
16
17    // add three circles to snowman, add snowman to scene
18    @Override
19    public void setup() {
20        blueSnowman = new CompositeSprite();
21
22        OvalSprite head = new OvalSprite(0.3, 0.3);
23        head.setLocation(0, -0.25);
24        blueSnowman.addSprite(head);
25
26        OvalSprite middle = new OvalSprite(0.4, 0.4);
27        middle.setLocation(0.0, 0.0);
28        blueSnowman.addSprite(middle);
29
30        OvalSprite bottom = new OvalSprite(0.5, 0.5);
31        bottom.setLocation(0, 0.35);
32        blueSnowman.addSprite(bottom);
33
34        // snowman scaled and placed off center on screen
35        blueSnowman.setLocation(0.67, 0.67);
36        blueSnowman.setScale(0.33);
37        addSprite(blueSnowman);
38    }
39 }

```

Listing 4.2: Snowman demonstration program

The Snowman program is a demonstration program in the flavor of those presented in Chapter 2: light on comments, short, and complete. Notice that in setup addSprite is called four times: three times on blueSnowman and once on the game (with blueSnowman as the parameter). The locations of the three snowballs are inside the CompositeSprite.

To demonstrate that the three OvalSprites are treated as a single unit, the program rotates the snowman about 4 times a minute. The screenshot for this program is of a slightly modified version of the program, one which adds white lines as the x and y axes of the CompositeSprite. This is so you can easily spot the center of the snowman.

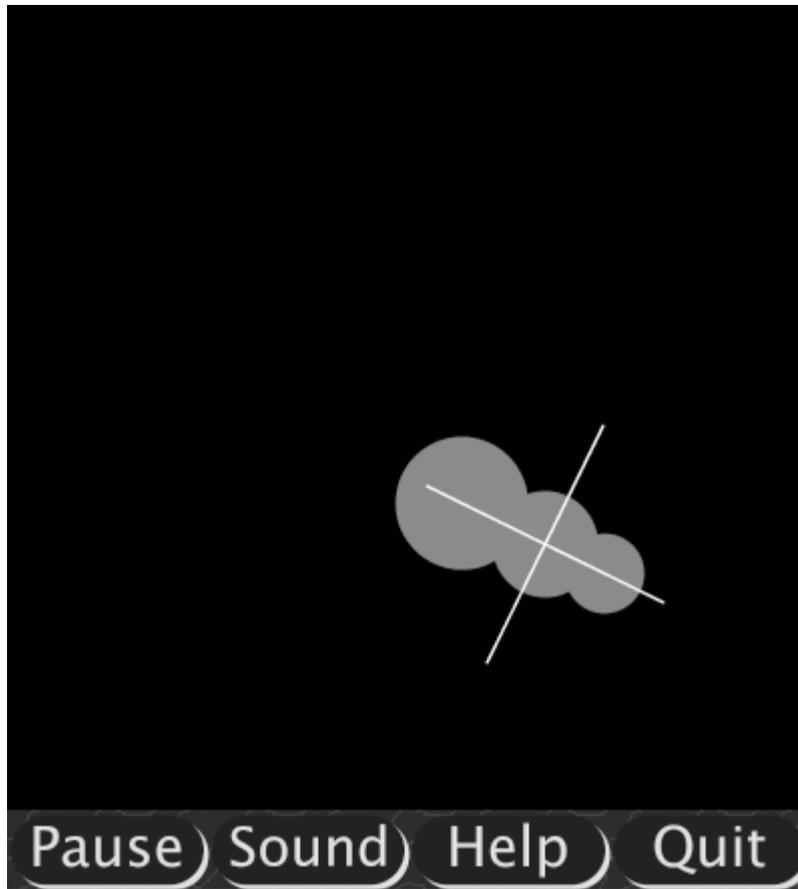


Figure 4.4: Running AxisSnowman

Button, Button, Defining a Button

As demonstrated above, it is possible to define a `CompositeSprite` on the fly, in the `setup` method of a `Game` extending class. This is useful if you have need of exactly one of some kind of grouped sprite and its only state is visual (where it is, its scale, and its rotation). If you might want more than one snowman or if you want to keep additional information for the group, it is almost always better to extend `CompositeSprite`. This way you can put all the sprite creation and adding in the constructor and you can make as many as you want, positioning, scaling, or what ever.

Doing this for the snowman is left as an exercise for the reader. We will, instead, shift our focus to the `EasyButton` class.

Writing a Constructor

There are four parts to the signature of a standard method: access level, return type, name, and parameter list. Then the signature is followed by a block containing the body of the method.

Until now we have not written a *constructor*. The signature of a constructor is different than other methods in two ways: its name must match the name of the class *exactly* and there is no return type. The first requirement makes sense when you look at how the constructor is used. Listing 4.1, line 26 shows the initialization of button:

```

37 button = new EasyButton();
38 button.setScale(0.5);

```

```

39     button.setLocation(0.5, 0.85);
40     button.setColor(getColor("yellow"));
41     button.setTextColor(getColor("navy"));
42     addSprite(button);

```

Listing 4.3: RollDice demonstration program

After `new` is a call to the constructor. It names the type that is being constructed. The second requirement, that there is no return type, is strange until you realize that a constructor is making the type that is named in its name.

```

9  /** A button class. Twice as wide as high with centered text message. */
10 public class EasyButton
11     extends CompositeSprite {
12     /** the button at the bottom of the screen */
13     private final RectangleSprite button;
14
15     /** text displayed, centered, on the button */
16     private final StringSprite buttonMessage;
17
18     /**
19      * Construct new button. Horizontal, FANGBlue with same color text.
20      */
21     public EasyButton() {
22         button = new RectangleSprite(1.0, 0.5);
23         addSprite(button);
24         buttonMessage = new StringSprite();
25         addSprite(buttonMessage);
26     }

```

Listing 4.4: EasyButton: declaration and constructor

This listing shows the first portion of the `EasyButton` definition. Notice that this `class` extends `CompositeSprite`. This means it has all the functionality of a sprite (so `setLocation`, for example, works on objects of this type). The two fields, `button` and `buttonMessage`, refer to the `RectangleSprite` and the `StringSprite` after they are created. By keeping references to the sprites which make up the composite our code can implement `EasyButton.setText`, `EasyButton.setColor`, and `EasyButton.setTextColor`. Without a reference to the message, there is no way to set the text appearing on the button.

The constructor, lines 21-26, constructs two sprites, a rectangle and a text string. Both are positioned at (0.0, 0.0) *inside* the `CompositeSprite` and then they are both added using `addSprite`. Remember the layering order of sprites; the layering order inside a `CompositeSprite` is the same as it is in a regular screen so the rectangle is added before the text.

In this constructor we used `addSprite` with nothing in front of it; in setup above we used `blueSnowman.addSprite`. What is the difference? In setup, calling `addSprite` alone will add to the current object, an instance of the Game-derived `Snowman` class. The sprite would be added to the screen, not the composite. Similarly, just `addSprite` in `EasyButton` adds the sprite to the current object, an instance of the `CompositeSprite`-derived `EasyButton` class rather than to the screen. Line 42 of Listing 4.1 adds a new `EasyButton` to the game.

John von Neumann (duplicate) [1903-1957]

John von Neumann was a prodigious mathematician who contributed to the development of game theory and computer science. He was born in Budapest, Hungary just after the turn of the Twentieth Century, received his PhD in mathematics by age 22, and emigrated to the United States in 1930.

John von Neumann (contd.)

In the US, von Neumann was an original member of the Institute for Advanced Study at Princeton University, one of the first places in the world to build digital electromechanical and electronic computers. He also contributed to the computing power, mathematics, and physics of the Manhattan Project, the United States' World War II super secret initiative to develop the worlds first atomic bomb.

While consulting on the Electronic Discrete Variable Automatic Computer (EDVAC), von Neumann wrote an incomplete but widely circulated report describing a computer architecture which used a unified memory to store both data and instructions. This contrasted with the so called *Harvard* architecture where the program and the data on which the program operated were segregated.

The von Neumann architecture is the fundamental architecture used in modern computers. A single, unified memory makes it possible for a computer program to treat itself (or another computer program) as data for input or output. This is just what the operating system described in this section or the compiler describe in the next section does.

Von Neumann's report drew on but failed to acknowledge the work of J. Presper Eckert and John W. Mauchly; this led to some acrimony when it came time to allocate credit for the modern computer revolution.

Von Neumann's name will come up again when we examine game theory, a branch of economics/-mathematics that permits reasoning about strategies in competitive games.

Modifying State and Overriding Methods

`EasyButton` is a `CompositeSprite` in the same way that a ham sandwich is a sandwich. Anything you can do with a generic sandwich you can do with the more specialized **ham** sandwich. Of course there are things you can do with a ham sandwich, such as get the ham, which are not possible for arbitrary sandwiches. The following code shows several methods for updating the colors and text of an `EasyButton`.

```

44  /**
45   * Set the color of the background rectangle
46   *
47   * @param color the color to set the rectangle to.
48   */
49  @Override
50  public void setColor(Color color) {
51      button.setColor(color);
52  }
53
54  /**
55   * Set text message; resize to fit in one line
56   *
57   * @param message the new message for the button to display
58   */
59  public void setText(String message) {
60      buttonMessage.setText(message);
61      // adjust size of text message to fit on the button
62      buttonMessage.setWidth(0.75);
63  }
64
65  /**
66   * Set the color of the text on the button.
67   *
68   * @param color the color to set the text to
69   */

```

```

70 public void setTextColor(Color color) {
71     buttonMessage.setColor(color);
72 }

```

Listing 4.5: EasyButton: set methods

Lines 59-63, the `setText` method, makes use of the identically named method of `StringSprite`. The `String` parameter (a type for holding sequences of characters) is passed directly to `buttonMessage`'s `setText` method. Then the scale of the message is reset so that its width fits in the middle 75% of the displayed button.

A `CompositeSprite` is also a `Sprite` so it has a `setColor` method. Unfortunately for us, in `EasyButton`, the built-in version does not set the color of any of the elements within the composite sprite. We want `setColor` to set the color of the background rectangle (the `button` field).

Lines 49-52 *override* the built-in version of `setColor`. A method is overridden when a more specialized class provides a method with the same signature. The `@Override` on line 49 is called an *annotation*. An annotation can be thought of as a specialized type of comment, a comment that is processed by the compiler.

As a comment, line 49 tells the programmer that this is a specialized version of `setColor`, replacing the version provided above in `Sprite` (or, if that one was overridden, the one in `CompositeSprite`). The compiler reads the annotation and, for `@Override`, it makes sure that there is a method higher up in the class hierarchy to be overridden. If there is no such method, the `@Override` annotation throws an error. There are other annotations but they will not be discussed in this book; they are recognized by beginning with the `@` symbol.

The local `setColor` method sets the color of the rectangle to the `Color` parameter passed into it. This is similar to the `setText` method above as it forwards the work to the appropriate sprite within the composite. The `Color` type is a Java-defined type; importing `java.awt.Color` is necessary so that this method compiles.

Lines 70-72 are almost identical to `setColor` except that they set the color of the `StringSprite` within the composite. Notice that this method does *not* override another, identically named method. The parameter passed to `setTextColor` is forwarded to the `setColor` method of the `buttonMessage` field. These two color setting methods show why we kept references to the two sprites in the composite.

Game. getCurrentGame()

Looking back at `NewtonsApple` you can see how we used the `getMouse2D` method to determine where the mouse was during a given frame. Since we did not define it and it is not called with a variable and a dot, the method must be defined in `Game` and we get it for “free” when we extend `Game`.

One important thing to remember about `getMouse2D` is what value we get back if there is no valid mouse position. It is the special value `null`. Keep that in mind as we proceed with this section.

What does the `EasyButton`'s `isPressed` method have to do. This is important because `isPressed` is the last method in the public interface for `EasyButton` left to define.

The `isPressed` method returns `true` if the player has clicked the mouse this turn *and* they clicked it inside our button; otherwise the method returns `false`. How can we determine whether or not the player clicked the mouse? And how can we determine where the mouse was pressed?

```

33 public boolean isPressed() {
34     // The current game may have a mouse click or not.
35     Location2D mouseClick = Game.getCurrentGame().getClick2D();
36     if (mouseClick != null) {
37         if (intersects(mouseClick)) {
38             return true;
39         }
40     }
41     return false;
42 }

```

Listing 4.6: EasyButton: isPressed

Section 4.4, later in this chapter, explains how to use the Java and FANG documentation to examine the public interfaces of all of the various components each supplies. For now, suffice it to say, the `Game` class supplies mouse checking methods. In Section 3.4 we used `getMouse2D` to have `newton` follow the player's mouse.

In addition to `getMouse2D` which returns the location where the mouse is every frame (if it is within the game window), there is also `getClick2D` which returns the location where the mouse *clicked* during any frame where it was clicked. The location is encapsulated in the `Location2D` type; we will use `Location2D` through its public interface and leave the details until later.

If the player clicked any mouse button since the last call to `advance`, this method will return the location of the mouse click. If the player hasn't clicked any mouse button *or* the mouse is not in the window at all, this method returns... Any guess as to what it returns if there is no location where a click took place?

It returns `null` to indicate that there is no mouse click available. Thus `isPressed` should start by calling `getClick2D` and testing (using `selection`) whether the value is `null`.

Do you see the problem here? Inside a `Game`-derived class `getClick2D` is available; `isPressed` is inside of a `CompositeSprite`-derived class. We need to get a handle on the current game. Fortunately the `Game` class itself provides us with a utility method called `getCurrentGame` which takes no parameters and returns a reference to the currently running game. We can call `getClick2D` on that reference.

That is just what line 69 does, assigning the result to the local variable `mouseClick`. Whenever we need access to the current game from a sprite, be it for getting a color, getting a random number, or getting player input, we will use `Game.getCurrentGame`.

We want to return a value from this method; that is what the `return` statements, lines 38 and 41, do. the `return` statement's template is:

```
<returnStmt> := return <expression>;
```

Here the expression after `return` is evaluated and that value is returned as the result of the method. We want to return `true` if the mouse was clicked (`mouseClick != null` and, if the mouse was pressed, the mouse click location intersects the button (the button was clicked *on*). In addition to being able to test for intersection between two sprites, we can test for intersection of a location with a sprite. Thus `intersects(mouseClick)` in line 71 is a Boolean expression returning `true` if the click is inside the `CompositeSprite`. Note that the click and the sprite's boundaries are in *screen* coordinates (you must change the location to internal coordinates if you want to test intersection with subsprites).

When Java executes `return`, the expression is evaluated and the execution of the method halts (no other lines are executed). Thus if line 38 executes, line 42 cannot execute unless `ifPressed` is called again. We don't need an `else` in this case. This makes the routine a couple of lines shorter but still clear. Only if *both* `if` statement conditions are `true` does the method return `true`. If either is false, the method executes line 41 and returns `false`.

void methods If a method is of type `void`, `return` has the form

```
<returnStmt> := return;
```

This is because there is no expression of type `void`. All that happens when this `return` statement is run is that the method immediately returns control to the caller. No value is ever returned from a `void` method.

Object Types: `class`

Object types, the types with capitalized names, have played a major role in our sample programs so far. When discussing variables as labels for underlying components and the sharing of one component by multiple variables applies entirely and exclusively to *object* types.

- Objects must be instantiated (created). This is done using `new` or by letting the system construct an object for us inside of some specified method call.
- An object variable name is a label for an underlying object in the computer memory.
 - An object variable may refer to no object at all: `null`.

- An object variable may refer to at most one object in memory.
- What object a variable refers to can be changed through assignment.
- More than one label can refer to the same underlying object in memory.

Having these features firmly in mind, we now examine the primitive types and can see how the two types of types differ.

Plain Old Data: `int`, `double`, `char`

Not everything in Java is an object⁴; some components are so simple that Java provides facilities to manipulate them directly without putting them into an object. These simple components, components with *primitive types* are not instantiated, have no “null” value, and variables of this type never refer to the same underlying component. The primitive types are much more like boxes that can hold values than labels that can go on objects.

Java has several primitive types; they are easily distinguished from object type names in that all eight have names that begin with lower-case letters: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. The first four refer to slightly different types of whole numbers or *integers*; the next two refer to slightly different types of “real” numbers or numbers with a fractional part⁵; `char` refers to individual characters; and `boolean` refers to what we have been calling truth values, values that are either `true` or `false`.

Primitive Variables

Primitive types can be used to declare variables using the same syntax as object types. For example, the following line declares two `int` variables, `p` and `q`:

```
int p;
int q;
```

When a plain old data type is used to declare a variable, Java sets aside some space in memory to hold the value. It then permits you to refer to that location or “memory box” by the name of the variable. Notice that by declaring the variable’s type you have provided Java with the context necessary to interpret values stored in that box correctly.

The types `byte`, `short`, `int`, and `long` are all types that can hold integral numeric values. They differ in the range of numbers they can hold. `byte` uses eight *bits* (binary digits) and can hold values from $2^7 \dots 2^7 - 1$, inclusive. `short` uses sixteen bits to hold values from $2^{15} \dots 2^{15} - 1$, inclusive. `int` uses thirty two bits to hold values from $2^{31} \dots 2^{31} - 1$, inclusive (the numbers in the footnote on integer literals). `long` uses sixty four bits to hold values from $2^{63} \dots 2^{63} - 1$, inclusive. We will not worry too much about these distinctions, sticking with `int` values because they will encompass all of the ranges we want to use.

Assigning a value to an `int` variable uses the same syntax as assigning a value to an object-typed variable. It can be done on the line when the variable is declared or later. Assuming the following code is inside a method declaration, the following code assigns the value 99 to `p` and 121 to `q`:

```
int p = 99;
int q;

q = 121;
p = q;
```

⁴This is not true of all object-oriented languages. In Squeak, for example, everything is an object from the number 9 to the character ‘Q’ to the mathematical constant *Pi*. Java is a hybrid language with both objects and primitive types. Primitive types tend to improve both the speed and size of computer programs.

⁵“Real” is in quotes because computers cannot store the infinite values of actual real numbers; as explained later, these numbers are really a subset of the rational numbers.

When a new value is assigned to a primitive variable, the old value is destroyed. That is, in the last line above, right before the assignment executes, the value in the box labeled `p` is 99. Right after the assignment, the value is 121. The old value, 99, cannot be recovered from the variable `p`. Note also that the value 121 appears *twice* in memory, once in box `p` and once in box `q`.

The following code creates two integer variables and assigns a value to each of them and then changes one of the values:

```

1  int p = 99;
2  int q;
3
4  q = p;
5  p = 1001;

```

At the end of this code, what is the value of `q`? If `p` and `q` were *labels* for an underlying integer object in memory, one would expect the value to be 1001; since we were warned above that they are *not* labels for one location but rather independent values, it seems likely that the value is 99.

Similar declarations and assignment (using literals) can be done with the other primitive types:

```

double d = 1.345;
char ch = 'A';
boolean b = true;

```

The last example indicates that the keyword `true` (and, by implication, `false`) is a `boolean` literal, a value typed directly into the program. The type for storing truth values is called `boolean` in honor of English mathematician John Boole (1815-1864) who explored mathematics with only two values.

Literal Values

Primitive type values are not created using `new`. They are so simple that it is possible to include the value of a primitive type directly in the text of a computer program. For example, 100 is an example of a *literal* integer value. If the three characters “100” appear in an appropriate place in a Java program, they are interpreted as the integer value that comes after ninety nine (and before one hundred one). This interpretation is fundamental to Java and is always available.

An integer literal is an optional sign character followed by a sequence of digits. Examples include 100, -1, +3200000. Notice that the last example does not use separators to group digits (that is, it is *not* written 3,200,000). Integer literals are, by default, of type `int`, a type that can store numbers from minus two billion to plus two billion⁶. When writing an integer literal, avoid using any leading zeros as this changes the meaning of the number⁷.

A *floating point* literal, one that represents a “real” number, consists of an optional sign, a sequence of digits, a decimal point, and a sequence of digits. They are called “floating point” because the number of digits before and after the decimal point can vary (the point floats within the maximum number of digits that such a number can hold). Examples of floating point literals include 0.5, -19.876, and 100.0; notice that the part after the decimal can have the value of 0. Floating point literals are, by default, of type `double`⁸. Note that Java literals are typed; though 100 and 100.0 are both numeric literals and both represent the number one more than 99, they are *not* the same because one has type `int` and the other has type `double`.

A character literal represents a single character and is typed using single quotes. Examples include `'$'`, `'A'`, and `'n'`. The first represents the dollar sign character and the second the capital letter A (which is distinct from `'a'`, the character representing the lower case a). The third example is an example of a special *escaped* character. The backslash (`\`) is not itself considered the character but instead indicates that the next character indicates the value of the character. `'n'` represents the new line character.

⁶actually from -2147483648...2147483647 inclusive

⁷A literal that begins with 0 is considered to be in base eight (*octal*) notation; each character represents a power of eight rather than a power of ten.

⁸The range of a floating point type makes little sense to the beginner but a `double` can store about 16 digits of precision and values on the range of 10^{-1075} to 10^{971} .

4.3 Java Components

In a game, every component has a type. The type of the component determines what rules apply to that component. In chess, for example, a queen is one type of component, a king is another type of component. How each can move is determined by the type of the component. How the game unfolds after a piece is captured is also determined by the type of the component captured. In addition to a type, components also have attributes. Both the black queen and the white queen are queens; they both follow the same rules. Each has a different appearance, so the color is an attribute of the queen. The starting square and current square are also attributes of each of the two queens (color, starting square, and current square are attributes common to *all* chess pieces, including the queens).

In order to actually move your queen, it is necessary that you be able to find the piece. That is, while the queen is a component with defined rules and attributes, in order to apply the rules, you need to be able to refer directly to the piece. If, for some reason, you could not find the queen, you could not move it.

Note that it is possible to describe what a queen can do without actually having any actual, physical, queen piece. All of this discussion applies, by analogy, to components in Java programs.

In Java, every component has a *type*. A type defines a list of rules and attributes that each component of that type has. The list of rules or, in computer science terminology, *services*, that the type defines determines what components of that type can do. The list of attributes, or *fields*, are special values, such as color, that apply only to single components of a given type. Finally, a component can, optionally, have one or more references to it. A *reference* is a way of finding the object; we usually consider a reference to be a name which we can use to refer to the component (hence the name “reference”). The type of the object determines *what* it can be asked to do; a reference to the object determines *how* we ask it to do something.

Classes of Components: Types

One of the biggest differences between different programming languages is the types of components each language permits you to discuss. Many older languages such as Fortran and Cobol were designed to work in a single domain of components (Fortran, short for “*Formula Translator*”, was designed for scientific and mathematical programming and COBOL, “*Common Business Oriented Language*”, was designed for working with fixed-record databases).

Newer languages including Java take a more general approach by providing a set of built-in types *and* a mechanism for defining new types. Java’s *object-oriented* approach means that most components in Java are *objects* of some type defined outside of the core language. Any component created with the `new` keyword (as with the `OvalSprite` in the last chapter) or, by convention, any type name beginning with a capital letter, is an object⁹.

Java defines many different types in its standard libraries: `Color` and `String`, for example (notice the capital letters at the beginning of the names of these object types). FANG defines some other types: `StringSprite`, `Game`, and `RectangleSprite`, for example. Java permits programmers to define their own types using `class` as a building block; you have already defined your own types, one for each sample program you have written. As the examples and exercises become more complex, our computer games will use multiple programmer-defined types such as `Rook` and `Knight` for a Java chess game.

A user-defined type is a `class`; as we have seen, a class is a named container of attributes (fields) and rules (methods). *Fields* are attributes; they represent information about a particular object. The information stored is uniform across all objects of this type but the value of the information stored can differ. Consider the current square where a `Knight` is. Each of the four `Knight` objects will be on a different square (so the value stored is different) but *all* `Knight` objects need to keep track of that information.

Methods are rules; they define the actions that objects of this type can perform. The actual results of applying a given rule might depend on attribute values stored by the object, but all objects of a given type share the same set of rules. Going back to our four `Knights`, consider “move up one and left two” as a rule to apply. Each `Knight`, starting on a different square, will end up on a different square if this rule is applied.

⁹Java types come in two flavors: *plain-old-data* or *primitive* types that are defined in the base language and represent things like numbers and individual characters; and *objects* that are types defined in a library or in a user’s program. More on primitive types will be found later in this chapter.

Further, some of them might not be allowed to apply the rule at the moment (target square might be off the board or occupied by a friendly piece). But the exact same set of rules applies to each component of the type Knight (that is, any *other* Knight in the same position would have the same restrictions applied).

A Chess Example

The type of a component specifies what rules that component follows; it specifies a contract, the public interface presented by components of that type. Taking a broader view of the game of chess, the components are a game board of 64-alternating colored squares and 32 pieces of six different types (pawn, knight, bishop, rook, queen, king) in two different colors. The type of a piece determines what rules apply to that piece: a knight and a rook can both move but the knight can jump from one board square to another without regard to the intervening squares; the rook can, under certain circumstances, participate in castling. The exact type of the component determines the rules that it obeys; the rules a component obeys determine what it can do.

In chess, each component corresponds to a physical object. That is, each actual component with which the game is played is a separate physical object. There are thirty two different pieces in a chess set. In a programming language like Java, components correspond not to physical objects but rather to *logical* object that are created inside the computer's memory (more on what this means in the next section).

Types are not Instances

The type of a component is different from any given component of that type. The rules for Knight pieces is different than the white, King's Knight. In Java language terms, we say the *type* or *class* of an *object* is different than the object. An object is also known as an *instance* of its type. Thus `OvalSprite` is a type and, in `NewtonsApple`, `apple` refers to an instance of that type. The `apple` object is of type `OvalSprite` but the object and the type are two different things.

The distinction between a type and its object is similar to that between a blueprint for a building and a building built according to the blueprint.

Consider a subdivision, Samesville, where all homes are built from one blueprint. If you know the blueprint you could get from the front door of any house to that house's kitchen and find the refrigerator (each instance of the blueprints supports the same `getRefrigerator()` method). It would work at 99 Midland Rd. or at 123 Brockport Ln. The process of putting an orange into the refrigerator (`addToRefrigeratorContents()`) is also uniform across all the matching houses. Yet, only the actual house where you put the orange into the refrigerator will have your orange.

The blueprint defines `addToRefrigeratorContents()` method for all houses, each house has its own refrigerator. The refrigerator and, in particular, its contents are an *attribute* of each particular house. You put an orange into the refrigerator at 123 Brockport Ln. Thus, while the type (blueprint) defines `getRefrigeratorContents()` for all houses, the orange is only available from that one particular house. Notice that each house has an address, a label that permits you to differentiate between them. Thus you could write something like the following to put the orange away:

```
1 BrockportLn123.addToRefrigeratorContents(myOrange); // Fake Java
```

(The address is reversed so that it mimics a valid Java label; more on acceptable Java names follows later in the chapter.)

A reference to a component is a *name* that we can use to refer to the component. To refer to the methods of a given component, to ask that component to follow one of its named rules, we use the *dot notation*: a name referring to the component followed by a dot (".") followed by the name of the method and a parameter list. In the fake Java above, the parts of the method call are:

reference `BrockportLn123`

dot `.`

method name `addToRefrigeratorContents`

parameter list `(myOrange)`

Declaring a Name

What is in a name? Would an object by any other name not compute as fast?¹⁰ A name in Java is a label for an object (later we will see what happens with non-object components). At any given time a name refers to at most one component (it can refer to *no component* with the `null` value). Java is a *strongly-typed language* which means that each name has an associated type and a name can only be used to label objects of appropriate types. While this seems limiting, it permits the compiler to catch many typographic and logical errors that would be much harder to diagnose if the program were permitted to compile and run before there was a problem.

A Java name, or *identifier*, is a sequence of characters that begins with a letter or an underscore and continues with zero or more letters, underscores, or digits. Thus an identifier is one or more characters long and starts with a letter or an underscore. Examples of valid identifiers include:

- `getX`
- `RectangleSprite`
- `NCC1701D`
- `ALEPH0`
- `D1999_Fall`
- `_someIdentifier`
- `smithfield`
- `Smithfield`
- `SMITHfield`

Notice that the last three identifiers are different: Java is *case-sensitive* and upper and lowercase letters are not the same. Examples of invalid identifiers (and why each is invalid) include:

- `3Dimensional` — begins with a digit, not a letter or an underscore
- `the#ofGoodies` — contains an invalid character, “#”
- `my first name` — contains spaces (an invalid character in identifiers)
- `class` — “class” is a Java *keyword*, an identifier reserved for the use of the language. A complete list of Java keywords is in Appendix A

Names have Types

Component labels are *declared* as *variables*. As we saw in the previous chapter, a field declaration is

```
<fieldDeclaration> := private <TypeName> <fieldName>;
```

where the type and the name are replaced with an appropriate type and an appropriate name. For example, inside of `OneDie` we will need to keep track of the current value of the die’s top face. This would be an `int` so the field would be declared:

```
private int face;
```

Similarly, above in `EasyButton` when we declared the fields for the rectangle and string sprite the declarations were

```
private RectangleSprite button;
private StringSprite buttonMessage;
```

¹⁰Apologies to the Bard

Names are Labels

Whenever you want to create an object of a given type you use the `new` keyword to tell Java to set aside space to hold all the attributes of an instance of the object you are creating. You must provide the name of the type you want to create so Java has a plan to build from. The running program looks up the named type to determine how much memory it needs. It also calls a special method called a *constructor*. The constructor initializes all of the fields of the object so that it is ready to use.

Each instance of a type has its own copy of all fields; all of the instances share the same methods. This is similar to our chess analogy: each of the sixteen pawns (each an instance of the type `Pawn`) starts the game with a specific color and specific board location (the fields) but every pawn follows the same rules for moving, capturing, and being captured (the methods; note that “move one square toward the opponent’s rear rank” requires knowing the color of the pawn as well as the pawn’s current position. These are, exactly, the fields each instance of the type maintains).

Each call to `new` with the same type name creates a distinct instance of that type. The following lines create three different rectangles that can each be added to a `Game`:

```
1 new RectangleSprite(0.10, 0.10);  
2 new RectangleSprite(0.20, 0.20);  
3 new RectangleSprite(0.40, 0.40);
```

These three `RectangleSprites` are one tenth, two tenths, and four tenths of a screen in size (each is a square). Every `RectangleSprite` supports the same interface, that is, can do the same things; each of these three rectangles is an independent instance. Setting the color of the small rectangle to red will *not* change the color of the other two rectangles. Setting the color of the small rectangle does bring out a particular problem: how can we change the color of any of these rectangles? Knowing the interface that an object supports is half of the information we need to ask the object to do something. We also need some way to refer to the specific object we want to manipulate; we need a *name* for the object.

In Java all variables with a class type (typically, with a type name that begins with a capital letter; see Section 4.3 for more on naming conventions) are *references* to an object of that type. They can also be thought of as *labels* for such objects.

What does this mean? It means that a variable of type `RectangleSprite` is a label that can refer to any given `RectangleSprite` or, if we wish, to no `RectangleSprite` at all. It also means that any given `RectangleSprite` can sport any arbitrary number of labels at the same time. There is no way of knowing, given any one label, how many labels there are on at given object.

Analogy time again: Imagine a Java object is a car. Then the `class` definition is the blueprint for building the car and calling `new` has a new car constructed and returned. As with `new` in Java, if we wanted to refer to the car returned, we would need some label. One convenient label would be a license plate; you could assign the car to a given license plate and then refer to it through the license plate from then on. Note that while against the law, our analogy permits an arbitrary number of license plates per vehicle.

In Section 2.2 we saw that the computer’s RAM can be considered a long sequence of memory locations, each with its own address. Further, in that section and those surrounding it, we saw that the meaning of contents of memory depend on the type of value encoded into that memory location. This is important here because declaring a variable associates the variable with a specific memory location *and* determines the type of value encoded into a memory location.

The boxes in memory can only contain one value at a time. Whenever a value is assigned into a memory location, whatever was in the location previously is destroyed. When a memory location holds a reference to an object, assigning a different reference to the variable (storing the reference in the memory location associated with the variable’s name) *destroys* the previous reference but it does nothing to the object referred to.

With the license plate analogy, a license plate refers to at most one car at a time and changing the car which the license plate labels removes any connection between the license plate and its previously associated car.

Thus assignment of an object to an object variable is setting the variable to refer to the given object.

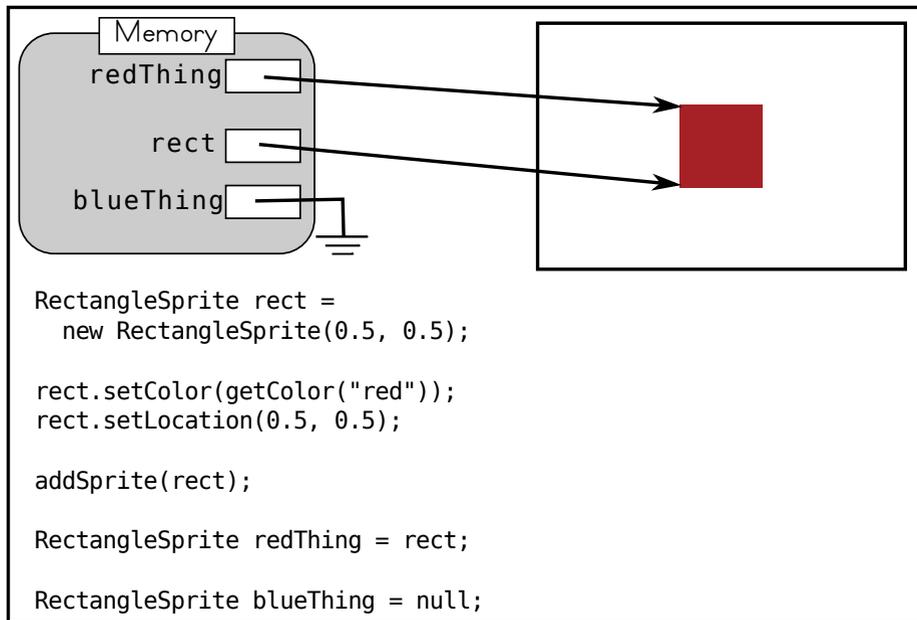


Figure 4.5: Illustrating Java references

Figure 4.5 illustrates how object variables work. The area labeled “Memory”¹¹ has boxes that can hold values. The boxes are not drawn next to each other because that is not something we need to know. We just need three boxes, one labeled for each `RectangleSprite`.

Both `rect` and `redThing` refer to the same rectangle, the one in the center of the screen. `blueThing` refers to no object at all or `null`. The `null` reference is pictured as the electrical *ground* symbol indicating that there is nothing referred to.

Given the situation in the figure, what would change if the following line were executed:

```
redThing.setLocation(0.25, 0.25);
```

The rectangle would be moved from the center of the screen to the upper-left corner of the screen. Note that either of the references to the rectangle could have been used to move it.

What is the value of `x` after the following code snippet executes:

```
rect.setLocation(0.60, 0.9);
double x = redThing.getX();
```

The first line moves the rectangle again, down and to the right. Then the next line queries the x-coordinate of the rectangle. Since `redThing` refers to the rectangle that was moved (with `setLocation`) to the point (0.60, 0.90), `x` is assigned the value of 0.60.

Names have Scope

In Java, all variable declarations are inside of a *block* (between a pair of `{ }` curly braces). The block they are in determines the *scope* of the name, the range of the program where the name is visible. Because these declarations are preceded with the keyword `private`, they must be declared within the `class` block¹².

¹¹The handwriting font is used for the label to remind us that this is something like how it really works but this is a high-level, human view of memory. The model is good enough to use until taking a computer organization or architecture course.

¹²We tend to declare everything that we can as `private`. This is because `private` data can only be touched by *this class*, the class where the private field or method is declared. This means we can protect the integrity of the values since only methods *in this class* can change them.

In Java, all class-level declarations are visible throughout the class; these fields are in scope from the beginning of the `class` container to the end of the `class` container. No matter whether the three rectangle names above are declared at line 10 or line 100 (or line 1000, for that matter), they are visible to every method within the class where they are declared (even those declared before them). We will use the convention of declaring all fields before any methods in the code in this book; be aware that this is just a convention and is not required by the language.

Variables can be declared inside of *any* block. Those declared inside any block contained inside the class body (a method block or a block inside a statement inside a method) are known as *local variables*. For example in `EasyButton`:

```
33 public boolean isPressed() {
34     // The current game may have a mouse click or not.
35     Location2D mouseClick = Game.getCurrentGame().getClick2D();
```

Listing 4.7: `EasyButton`: `isPressed`

Declaring a local variable is declaring an attribute that is local to the method in which it appears. It is, by definition, `private` so no visibility keyword is permitted.

Every time `isPressed` is executed, `mouseClick` begins with no assigned value. When Java executes the declaration, previous calls to this method are no longer available. Thus `mouseClick` has no value. When this method finishes (when execution runs off the end of the final closing bracket or a `return` statement is executed), `mouseClick`, the name, is taken off of any object to which it was attached and the name goes away¹³.

Local variable scope means that any *previously* assigned value (assigned during a previous call to `isPressed`, for example) was disassociated from the named label at the end of the previous call. This is where we see the “household atomic replicator” at work; when we left the scope of the variable the memory was reclaimed and we must explicitly request more memory for an object for it to refer to.

The Java compiler is acutely aware of uninitialized local variables, so much so that any attempt to use one will trigger a compiler error as in the following code:

```
70 public boolean isPressed() {
71     // The current game may have a mouse click or not.
72     Location2D mouseClick;
73     // vvvvv Compiler Error vvvvv
74     if (mouseClick.getX())
75     // ^^^^^ Compiler Error ^^^^^
76 }
```

Java is able to determine that between the declaration of the local variable `mouseClick` and `getX` call using `mouseClick`. (read as “`mouseClick` (dot)”), there is no way `mouseClick` was given a value. Thus the Java compiler can protect you from this kind of error. Note that it cannot always do this; class-level variables and parameters cannot be tracked like this.

Just like a class-level variable, a local variable can be assigned to as many times as you like. Assignment uses the single equals sign (=) which should be read as “is assigned” or “is assigned the value”. Local variables which refer to class types typically get their value from a call to `new` or some other method that creates a new instance of the given type.

Consider `ColorRectangle.java`, a program to create a randomly sized and randomly colored rectangle whenever and where ever the player clicks their mouse. Then the local variables in advance, `mouseClick` and `rectangle` are initialized between their declaration and the first time their value is used with comparison or a method call.

```
1 import fang.attributes.Location2D;
2 import fang.core.Game;
3 import fang.sprites.RectangleSprite;
```

¹³The *object* which was assigned to the variable does not, necessarily, go away. Remember that objects can have multiple labels so if there is another label referring to the object, Java keeps the *object* around.

```

4
5 /**
6  * ColoredRectangles demonstrates how to use local variables. The class
7  * has no fields and no setup method. In advance, it checks for a mouse
8  * click and if there is one, it creates a new, randomly sized and
9  * randomly colored rectangle centered where the mouse was clicked.
10 */
11 public class ColoredRectangles
12     extends Game {
13     /**
14      * Check for a mouse click; if one is clicked, make a random
15      * rectangle.
16      */
17     @Override
18     public void advance(double secondsSinceLastCall) {
19         Location2D mouseClicked;
20         mouseClicked = getClick2D();
21
22         if (mouseClicked != null) { // any click at all?
23             RectangleSprite rectangle;
24             rectangle = new RectangleSprite(randomDouble(), randomDouble());
25             rectangle.setColor(randomColor());
26             rectangle.setLocation(mouseClicked);
27             addSprite(rectangle);
28         }
29         // Is rectangle in scope?
30     }
31 }

```

Listing 4.8: ColoredRectangles

Notice the question on line 29. *Is rectangle visible at that point?* No, because a variable declared within a block is only usable within the block. `rectangle` is declared on line 23, the block it is in begins at the end of line 22, and the corresponding closing curly brace is on line 28. Thus `rectangle` is only in scope from line 23 to line 28; from where it was declared until the end of the closest enclosing block.

Java Naming Conventions

Identifiers are used to name methods, classes, fields, local variables, labels, parameters, and more. Naming conventions have developed to assist Java programmers reading code determine the kind of thing named. This section is derived from the Java standard naming conventions¹⁴.

Components: Named with Nouns

Variables, the names we are most concerned with in this chapter, should be in mixed upper and lower camel-case beginning with a lowercase character; as with methods, the lowercase initial letter is to indicate that the variable is contained *within* a `class` (or within a block within a class) rather than a class. Variable names should consist of a descriptive noun or noun phrase. Variable names can be declared directly in the scope of a `class` (an *instance* variable or a *field* of the class), in the *parameter list* of a method, or inside a method's body.

The amount of description required in a name is inversely proportional to the size of the scope of the variable. An instance variable holding the number of keystrokes entered by the user might be named `keys`, `keysNum`, or `totalNumberOfKeystrokes`. The last name in the list reads as a phrase describing what the value in the number *means*.

¹⁴<http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>

The second choice seems reasonable until you go back to work on your code after a long weekend. It is hard to remember whether you used “Numb” or “Number” and was it “numbKeys” or the other way around? There are a lot of different components that could be named “keys”: the actual values typed in by the user, some collection of objects found in the game world and used to open doors, and even a count of the number of keys pressed. It makes sense, at the `class` level, to make sure you give a full description of the variable.

If the variable were declared inside a method named `getKeyboardStatistics`, then a shorter name would make sense: `keyCount`, `count`, or even, perhaps, `keys`. These are acceptable because the name is used within a limited context where other meanings of “key” are unlikely to confuse readers of the code.

Constants: Unchanging Values

The one exception to naming variables in camel-case is for a special kind of variable: the *named constant*. A named constant can be recognized because it is declared with two special keywords, `static final`. These variables should be named with all uppercase letters with underscores, “_”, used to separate the words. Named constants are named with nouns or noun phrases and are declared at the class level. Examples might include:

```
static final double ROAD_X = 0.10;
static final double SITTER_X = 0.20;
static final double HOUSE_X = 0.90;
```

These named constants represent the x-coordinates of the road, sitter, and house, respectively. When they appear in a `setLocation` parameter list, it should be obvious whether or not they are being used correctly.

Consistency in code makes your code easier for a programmer to read. Any programmer, including you, can tell at a glance whether a name names a component, a rule, or an attribute.

Rules: Named with Verbs

Methods, the rules defined inside a `class`, should be in mixed upper and lower camel-case, starting with a lowercase letter. The camel-case makes the name easier to read. Consider `setColor`: the capitalized letters make it obvious that this name consists of three words. Compare that with a couple of variations: `setcolor` and `SETFILLCOLOR`. The variants are more difficult to read, especially after some practice reading camel-case code. The lowercase initial letter indicates that this name is contained within a `class`, not, itself, a class name.

Methods should be named with descriptive verbs or verb phrases. Two naming conventions are that Boolean methods, methods which return a truth value, typically begin with `is` (think `isPressed` from `EasyButton`) and fields are accessed through *getters* and *setters*, methods named `get<FieldName>` and `set<FieldName>`. Note that the first character of the field name is capitalized to put the method name in camel-case.

Classes: Named with Adjective-noun Phrases

Component type names, the names of `class` or *type names* (as well as `interface` names) should be in mixed uppercase and lowercase letters, with each word inside the name capitalized¹⁵. We have seen examples in the names of our sample programs, `NewtonApple`, `FANG` defined types, `RectangleSprite`, `OvalSprite`, and standard Java types, `String`. Note that as `String` is a single word, there is no internal capitalization.

When selecting a name for a `class` use a descriptive noun (a person, place or thing) or noun phrase. This is because a type represents a kind of component and components are *things*. Translating traditional physical games into Java we might find types such as: `ChessPiece`, `SixSidedDie`, `PlayingCard`, `DeckOfCards`. Be consistent in number when naming things (singular or plural nouns); this text will use plural nouns for collections of components and singular nouns in all other case. This means that `DeckOfCards` should contain a collection of `PlayingCard` objects.

¹⁵mixed case with internal words capitalized is also known as *camel-case* because the resulting names seem to have humps in them: `CamelCaseIdentifier`, `DrumSet`, `WhisperingWind`.

Manipulating Components: Expressions

An *attribute*, as we are using the term, is a value associated with and contained in a component. The `applesDropped` and `applesCaught` variables, defined as *fields* of the `NewtonApple` class back in Chapter 2, are examples of attributes we have used.

When the player catches an apple, the value of `applesCaught` is changed by assigning the result of an *expression* to `applesCaught`:

```
applesCaught = applesCaught + 1; // another apple caught
```

An expression is a piece of code which returns a value. This usage comes from the mathematical meaning of expression as “symbol or a collection of symbols representing some quantity or a relationship between quantities” [AHD00]. An expression, then, is a collection of code symbols that evaluates to some value. Like attributes, expressions have *types* as well as values.

Components have *state*, the current value of all attributes that are part of the component. A `Pawn`, in a chess game, has a color and a position; at a particular moment, the state of the `Pawn` might be `{white, "A3"}`; taken together, these two attribute values completely determine what the `Pawn` can do¹⁶. This chapter focuses on expressions and attributes and introduces a class which extends a class other than `Game`; details on how classes are defined occupy the next two chapters.

Simple Expressions

An *attribute*, as we are using the term, is a value associated with and contained in a component. The `applesDropped` and `applesCaught` variables, defined as *fields* of the `NewtonApple` class back in Chapter 2, are examples of attributes we have used.

When the player catches an apple, the value of `applesCaught` is changed by assigning the result of an *expression* to `applesCaught`:

```
applesCaught = applesCaught + 1; // another apple caught
```

An expression is a piece of code which returns a value. This usage comes from the mathematical meaning of expression as “symbol or a collection of symbols representing some quantity or a relationship between quantities” [AHD00]. An expression, then, is a collection of code symbols that evaluates to some value. Like attributes, expressions have *types* as well as values.

Components have *state*, the current value of all attributes that are part of the component. A `Pawn`, in a chess game, has a color and a position; at a particular moment, the state of the `Pawn` might be `{white, "A3"}`; taken together, these two attribute values completely determine what the `Pawn` can do¹⁷. This chapter focuses on expressions and attributes and introduces a class which extends a class other than `Game`; details on how classes are defined occupy the next two chapters.

Arithmetic Expressions

One thing computers are very good at is arithmetic. A numeric expression can be built by combining simple numeric expressions with unary and binary operators. The type of a numeric expression depends on the type of all of the subexpressions it is composed of.

The *unary* arithmetic operators are `+` and `-`; either can come before an arithmetic expression. `-` results in an expression with the opposite sign of the following expression; `+` results in the same expression it is given.

The *binary* arithmetic operators include `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division. The following example expressions are followed by their type and value in a comment.

¹⁶Chess, because it is thousands of years old, has some arcane rules and a `Pawn` needs to know its position on the *previous* turn as well as the current turn to support *en passant* capture. To simplify discussion, we will ignore *en passant* capture as well as castling with the rook and king (it is necessary to know that neither piece participating in a capture has move previously).

¹⁷Chess, because it is thousands of years old, has some arcane rules and a `Pawn` needs to know its position on the *previous* turn as well as the current turn to support *en passant* capture. To simplify discussion, we will ignore *en passant* capture as well as castling with the rook and king (it is necessary to know that neither piece participating in a capture has move previously).

```

1 + 4           // (a) int,    5
12 - 100        // (b) int,   -88
1.3 + 7         // (c) double, 8.3
1.0 + 4.0       // (d) double, 5.0
3 * 5           // (e) int,    15
3 + 2 * 3       // (f) int,    9
5.0 / 2.0       // (g) double, 2.5
5 / 2           // (h) int,    2

```

Numeric types narrower than `int` are widened to `int` before they are used with arithmetic operators. If the two *operands*, the subexpressions of the operator expression, are *not* the same type, the narrower expression is coerced to the type of the wider expression before the operator is applied. Thus in (c) above, the `int` 7 is widened to the `double` 7.0 before the operator is applied. This is typical of how a Java rule follower evaluates a binary operator expression: both subexpressions are evaluated and then the operator is applied to the two values; any required type coercion occurs just before application of the operator.

Example (f) above is 9, *not* 15 because of the *precedence* of operators, the rules that govern the order of application of operations in a complex expression. As was the case in math class, multiplication and division operators have higher precedence than addition operators. Thus `3 + 2 * 3` is evaluated as if it were `3 + (2 * 3)`, not `(3 + 2) * 3`. The programmer can use parentheses (as in the explanation) to change the order of operators; subexpressions in parentheses are evaluated before applying the operator the parentheses are an operand for.

Examples (g) and (h) bring up a difference between how most people define division and how Java does it. Example (g) divides the `double` value for five by the `double` value for two and comes up with two and a half, the value most students expect. Example (h) shows how Java defines *integer* division. Since both sides of the operator are `int` expressions, Java returns the whole number of times the divisor goes into the dividend. Since 2 goes into 5 2 times with a remainder of 1, `5 / 2` evaluates to 2.

Java provides a related operator, `%`, called the *mod* (short for *modulus*) operator. The mod operator evaluates to the *remainder* when the divisor divides the dividend. That is `5 % 2` evaluates to 1 and `19 % 7` evaluates to 5. The mod operator is useful for determining whether a given value is a multiple of a given value; the remainder of a multiple of 11, when divided by 11, is 0 by definition. It is also useful for counting over a limited range for repeating things like the frames of an animation.

The basic assignment operator in Java is `=`. We have already used it when we needed to bind a label to an underlying object or copy a primitive value into a primitive variable. We have always used it in a statement (by placing a `;` after the expression) but assignment is an expression almost like any other in this section.

“Almost” because assignment is an example of an expression that does more than evaluate to a value. In addition to returning a value, evaluating an assignment expression also changes the value of the variable on the left-hand side of the assignment operator. An assignment operator has a *side-effect*.

Assignment expressions cannot be considered in isolation just using literal expressions; the left-hand side of the assignment operator must be a label or a primitive variable or an expression that evaluates to a label or primitive variable (we have not yet seen any such expressions). In our examples we will define a couple of variables and then give examples of assignment expressions using them.

```

int someInt;
double someDouble;
String someString;

someInt = 88 % 3           // (a) int, 1
someDouble = 4.5 * 600    // (b) double, 2700.0
someString = "alpha" + "bet" // (c) String, "alphabet"

```

Two things to note: these expressions return a typed value just like any other expression; the value returned from an assignment expression is the value just assigned. This means that you *can* chain assignment expressions together or use an assignment expression as one of the subexpressions in a comparison operator

(for example). *Don't!* Using an assignment expression (or any other expression with a side-effect) as a subexpression in a larger expression leads to missing the side-effect when reading the code. Failing to see what is going on in code you are reading leads to misunderstandings and errors. One easy way to make your code more readable is to avoid side-effecting subexpressions.

In many computer programs it is useful to modify a particular variable with its new value based on its old value. The variable would then appear on both the right-hand side of the assignment operator (where it would be evaluated for its value) and on the left-hand side of the assignment operator (where it provides a name where the result (or a reference to the result) should be stored). For example:

```
int x, y, z;
x = 1;
y = 2;
z = 33;

x = x + 1           // (a) int, 2 --- x is assigned 2
y = y * 19         // (b) int, 38 --- y is assigned 38
z = z % 5          // (c) int, 3 --- z is assigned 3
```

Because this operation is so common (we will see a lot of it in our loops later in this chapter) Java has *compound assignment operators*. Most arithmetical operators can be combined with an equal sign to make an operator that means “evaluate the right-hand side expression then combine it with the variable on the left-hand side using this operator”. For example, the assignment expressions above could be rewritten as:

```
x += 1             // (a') int, 2 --- x is assigned 2
y *= 19           // (b') int, 38 --- y is assigned 38
z %= 5            // (c') int, 3 --- z is assigned 3
```

The compound assignment operator, += is also overloaded for String:

```
String someString;

someString = "A moose";
// (a) String, "A moose" --- someString assigned value
someString += " on the";
// (b) String, "A moose on the" --- someString assigned value
someString += " loose.";
// (c) String, "A moose on the loose." --- someString assigned value
```

This can be useful when building up a String, only part of which is known at any given time.

Method Invocation

Some of the simple expressions above ask an object to perform some task. That is, a method is *invoked* on some object. This form of simple expression has four parts:

- An expression that resolves to a reference to an object.
- A period (.); also read as “dot”.
- The name of a method provided by the type of the object referred to.
- A parenthesized list of the *parameters* (if any) that the method requires. Each parameter is an expression that is evaluated *before* the method is called with the result of the expression assigned to the value of the local name of the parameter in the method body.

In the previous section's String examples, (b) and (c) both invoked methods that were both defined in String and returned something of type String. It is not necessary that a method return an object at all and if it does, it can return any type of object that the author wants.

Boolean Expressions

Comparison operators produce **boolean**, or truth values, depending on the relative values of their subexpressions. This is another way to get a *booleanExpression* as used in last chapter's selection statements.

One comparison is for equality: is *expression_a* equal to (or not equal to) *expression_b*? The `==` operator is the “test for equality” operator¹⁸. The `!=` operator is the inequality comparison operator.

While it is possible to use either equality or inequality tests on object-typed expressions as well as plain-old data expressions, it almost never means what you think it does. Thus in this book the equality and inequality operators will only be applied to plain-old data types¹⁹. For object types we will rely on the `equals` method defined in the type.

All comparison operators have lower precedence than all arithmetic operators. This means that each of the following expressions evaluates the arithmetic values on either side of the comparison operator *before* comparing the values:

```
10 * 2 + 3 < 124 % 100    // (a) 23 < 24; boolean, true
4.5 / 3 >= 3.0 / 2        // (b) 1.5 >= 1.5; boolean, true
1972 * 1.1 < 2169         // (c) 2169.2 < 2169.0; boolean, false
```

The type of any truth value expression is **boolean**. In Example (c), note that the type of the right-hand subexpression is widened to **double** before the comparison takes place.

Two character operators must be typed with no space between the two characters. That is, less than or equal to is typed `<=`; the sequence `<-=` is interpreted as the operator less than followed by the assignment operator (and, since that sequence of operators makes no sense, the Java compiler will complain).

Logical operators combine Boolean expressions into more complex truth values. The three standard logical operators are `!` (*not*), `&&` (*and*), and `||` (*or*). The truth value of a logical expression depends on the truth value of the expressions these operators are applied to. `!` inverts the truth value: **true** becomes **false** and vice versa²⁰. When two **boolean** expressions are combined with `&&`, the expression is **true** if *and only if* both expressions are **true**. When two **boolean** expressions are combined with `||`, the expression is **true** if either (or both) of the expressions are **true**.

```
(1 < 2) && (3 < 4)        // (a) boolean, true
(1 > 2) && (3 < 4)        // (b) boolean, false
(1 > 2) || (3 < 4)       // (c) boolean, true
(1 > 2) || (3 > 4)       // (d) boolean, false
!(1 > 2)                 // (e) boolean, true
```

Note that the parentheses are not necessary because of precedence. The following rewrite of Example (a) above evaluates in just the same way:

```
1 < 2 && 3 < 4           // (a') boolean, true
```

It should be clear that the Example (a) is more readable than Example (a'); for the cost of typing a few parentheses the readability is well worth it.

Object Construction

The one other type of expression is the result of using the **new** operator. It creates a new object (of the type specified by the constructor after **new**) and returns a reference to that value.

¹⁸Students are often confused by the assignment operator (`=`) and the compare equality operator (`==`). One way to make reading code clearer is to read assignment as the words “is set equal” and comparison as “test equal to”.

¹⁹Exception: All object references can be compared to the special value, **null**. We will use equality and inequality to check references for **null**.

²⁰Given that `!` is read as *not*, the use of `!=` for *not equal* should now make more sense.

Overloading Methods

Java programmers cannot overload operators; all operator definitions are specified in the language standard. It is possible, however, for Java programmers to *overload* method definitions. What does that mean, how is it done, and why would we want to?

Consider writing a method called `perimeter`. We will write the method inside of a `Game` extending class. How can we figure out the perimeter of a `Sprite`?

Well, we can figure it out for a `RectangleSprite` fairly easily:

```
public double perimeter(RectangleSprite r) {
    return 2 * r.getWidth() + 2 * r.getHeight();
}
```

This is just using the fact that an axis-aligned rectangle has a perimeter equal to double its height plus its width. Can we redefine this so it works for `OvalSprite` too? For the moment we will assume the `OvalSprite` is a circle (both diameters the same).

The formula given above is not particularly close to the right answer. The above would give 4 times the diameter of the circle when the right answer is π times the diameter. What if we wrote *two* different methods in the same `Game`:

```
public double perimeter(RectangleSprite r) {
    return 2 * r.getWidth() + 2 * r.getHeight();
}

public double perimeter(OvalSprite o) {
    return Math.PI * o.getWidth();
}
```

There are now two different definitions for the same method. Java can tell them apart because they differ in signature: the parameter lists are different. Java will match up the parameter types when compiling and call the right version:

```
RectangleSprite first = new RectangleSprite(1.0, 0.5);
RectangleSprite second = new Rectangle(0.25, 0.25);

OvalSprite third = new OvalSprite(0.25, 0.25);
OvalSprite fourth = new OvalSprite(1.0, 1.0);

double firstP = perimeter(first); // perimeter(RectangleSprite)
double secondP = perimeter(second); // perimeter(RectangleSprite)
double thirdP = perimeter(third); // perimeter(OvalSprite)
double fourthP = perimeter(fourth); // perimeter(OvalSprite)
```

Each call is commented with which version actually gets called. This is useful when there are different ways to handle different types of objects *and* the compiler knows that type they are at compile time. This is different from *overriding*: Java will call the lowest, most specifically defined overriding method for an object; Java will call the overloaded method which most closely matches the types of the parameters as declared in the source code.

4.4 Examining a Public Interface

As we have mentioned earlier, Java was designed with a small core language (the keywords and the primitive types are built-in) and an extensive collection of standard libraries. Java programmers need to be able to find the classes provided by the various libraries and the methods provided by the various classes so Java has extensive documentation for its standard libraries. As we will learn in later chapters, it also includes tools for documenting the classes and libraries that we write.

The Java documentation is provided as a collection of Web pages. These can be found at the Java Web site (<http://java.sun.com/javase/6/docs/api/>) or can be installed from Sun's download web pages. The following screen shots have the documentation installed locally in the default directory.

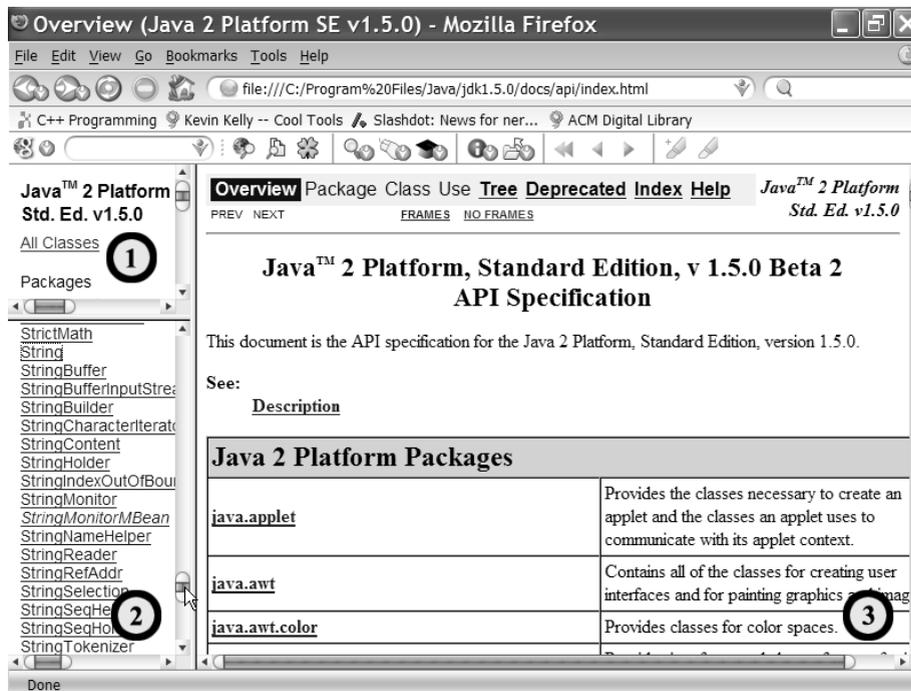


Figure 4.6: The initial Java documentation page.

Figure 4.6 shows the first page of the documentation with three panes:

1. *Packages Pane* Java collects classes into packages; this pane permits you to select a package to view. If you select a package, the classes in the *Classes Pane* will be limited to only those classes in the selected package. This is useful for browsing packages of related classes. You can always select *All Classes* to show all the classes in all of the libraries that Java knows about.
2. *Classes Pane* An alphabetical listing of all of the classes in the currently selected package (or all the classes in all packages if no package is selected). Searching in this pane permits you to find, for example, the entry for the `String` class.
3. *Documentation Pane* The documentation for the selected class or, if no class has yet been selected, a summary of the available packages. Note that the title of the package list includes “API Specification”. An *API* is an *application programmers interface*; this is a fancy way of talking about a standard collection of components and rules or, in computer speak, *data structures and interfaces*.

Finding Types and Public Interfaces

Figure 4.7 shows the result of clicking on the `String` entry in the *Classes Pane*. The documentation for the `String` class appears in the *Documentation Pane*. The key things to note are that above the `Class String` headline the package containing the class is listed: `java.lang` in this case. The `java.lang` package is the fundamental Java library and is imported automatically (no `import` line required); we will look at `Color` in a moment and see how to determine the `import` line(s) needed for a given class.

The class `String` is declared in its Java file with the lines

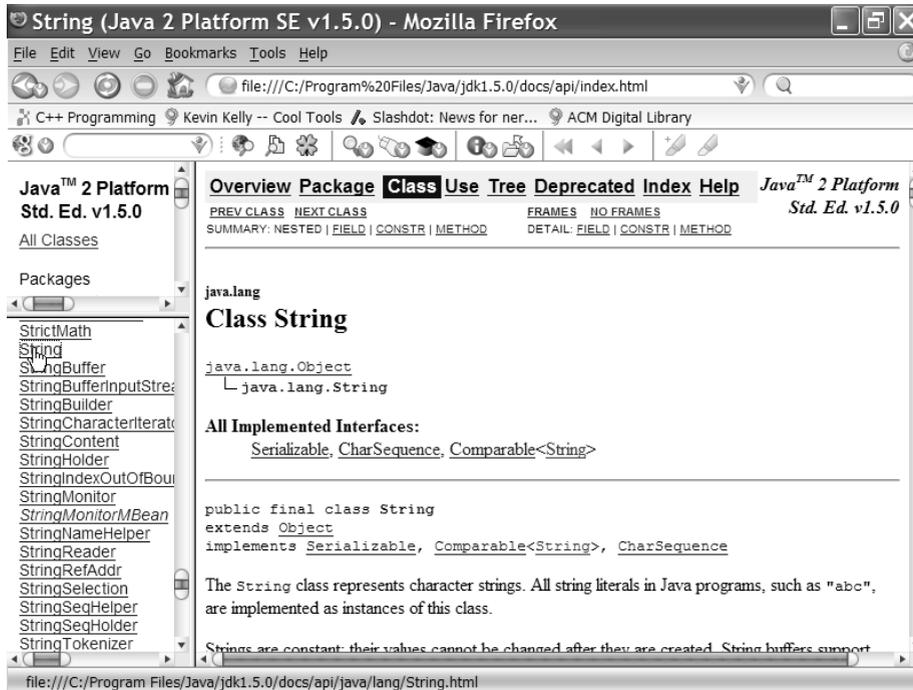


Figure 4.7: Top of the String class documentation.

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

which appear at the beginning of the documentation for the class; if we ignore **final** and the `implements` line, this is similar to the `class` declarations we have used for our sample programs. The `Object` class (the class that `String` extends) is *the* fundamental object type in Java; all object types extend it either directly (as here) or indirectly. Exactly what it means to extend a class is the subject of Chapter 6.

The text after the declaration of the class describes how the `String` class was intended to be used. Included just below the text in the screen shot is a mention that `Strings` are constant and a description of how the `+` operator is used for concatenation. Below the text is a series of tables: *Field Summary*, *Constructor Summary*, *Method Summary*.

The *Field Summary* lists the fields that are visible outside the class. We will look at how to set the visibility of fields when we study how to write our own classes.

The *Constructor Summary* lists all of the forms that can follow `new`. That is, in the `String` documentation, the following lines appear in the *Constructor Summary*:

```
String()
String(String original)
```

These are two (of several) constructors. These indicate that it is possible to use the following lines to create two different strings:

```
String firstString;
String secondString;

firstString = new String();
secondString = String("Alphabet Soup");
```

This code creates a new, empty `String` and labels it `firstString` (the fact that it is empty comes from the comments in the *Constructor Summary*). It then creates a new `String` that is a copy of "Alphabet Soup" and labels that `secondString`. Each constructor in the summary is listed with the parameters it takes and a one-line description of how it works. A more complete description of how it works is linked through the name of the constructor (each is linked to the description of a particular constructor).

The *Method Summary* is similar to the *Constructor Summary* in that it lists the name and parameter list of each method. Figure 4.8 shows a section of the *Method Summary* for the `String` class. The column on the left shows the *return type* of the method along with any special modifiers used in declaring the method (you can see the `static` modifier on the `copyValueOf` methods, for example; `static` methods will be explained when we write our own classes). The second column has the name and parameter list of the method. The parameter list tells you what types to call the method with. Below the name of the method is a one line description of what it does. One thing to keep an eye out for is the word *Deprecated*. A method that has been deprecated is no longer supported and may be removed from the library in the future; avoid using deprecated methods. A more detailed description is linked to the name of the method.

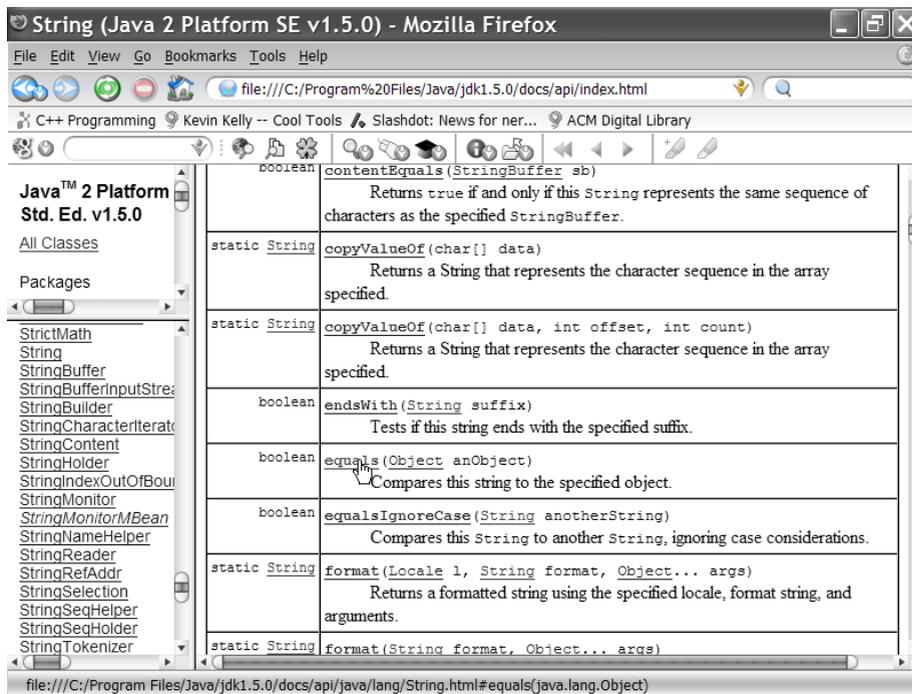


Figure 4.8: The methods of the `String` class.

Figure 4.9 shows the detailed explanation of the `equals` method. The return type of the method is `boolean` so it returns a truth value. The description says it returns `true` if and only if the parameter passed to the method is a `String` and contains the same characters as the object used to call the method. This is different from the `==` operator which is only `true` if both sides label the exact same underlying object.

Figure 4.10 shows the documentation page for the `Color` type. Again, note that the package containing the class appears right before the name of the class in the title. If this package is *not* `java.lang`, then you must import the class in order to use it in your program. Looking at this page, it is easy to see where the import line we have been used

```
import java.awt.Color;
```

comes from.

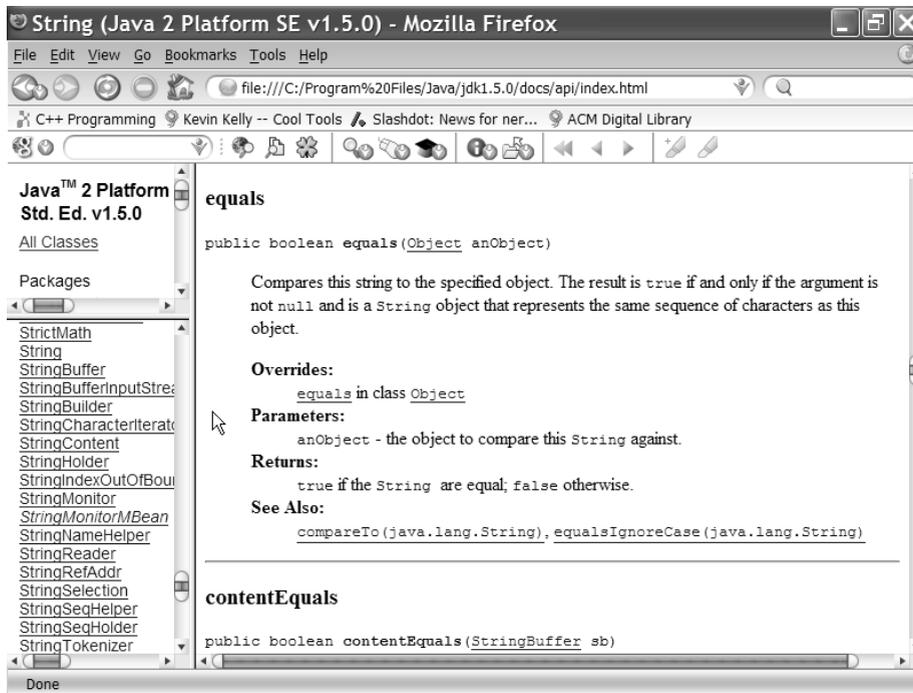


Figure 4.9: Documentation of equals method.

Java's documentation is very thorough and quite well linked (class names lead to the documentation page for that class). With upwards of 1500 classes, however, finding just the one you need can be daunting; reading package and class descriptions, even at random, can increase your grasp of these essential types.

JavaDoc: Two Audiences Again

When writing class and method header comments, you have probably wondered about some of the formatting. The template for a multi-line comment specifies it begins with `/*` and ends with `*/`, yet all of the header comments in sample code begin with `/**`. And then there are a lot of `@` signs peppered through the comments. What is all of that about?

That is all about JavaDoc. The Java development kit from Sun (and most other providers) includes `javac`, the Java compiler, `java`, the Java bytecode interpreter, and `javadoc`, the Java *documentation compiler*. The documentation presented in this section is automatically generated from the source code of the library classes.

The complete features of `javadoc` are beyond the scope of this book. Suffice it to say that all comments beginning `/**` are processed by the Java documentation compiler and the various `@` names are directions to the documentation compiler of how to format and link the resulting code.

This means that header comments, rather than just being written for fellow humans, are also written for two audiences: the programmers who come after you in the source code *and* the `javadoc` program.

This section will end by evaluating a quote from Norm Schryer, an AT&T researcher and computer scientist by way of an article called "Bumpersticker Computer Science" [Ben88]: "If the code and the comments disagree, then both are probably wrong." This is a great bumpersticker in that it is fairly short, pithy, and deeper than it seems.

The recommendation that you document your *intent* rather than your implementation makes it easier for the code to evolve without changing the commented behavior. The DRY Principle is about having one particular place where some action takes place or some value is set; this keeps you from having to worry about missing an update to either source or comments. Quality code requires constant vigilance against letting errors creep in.

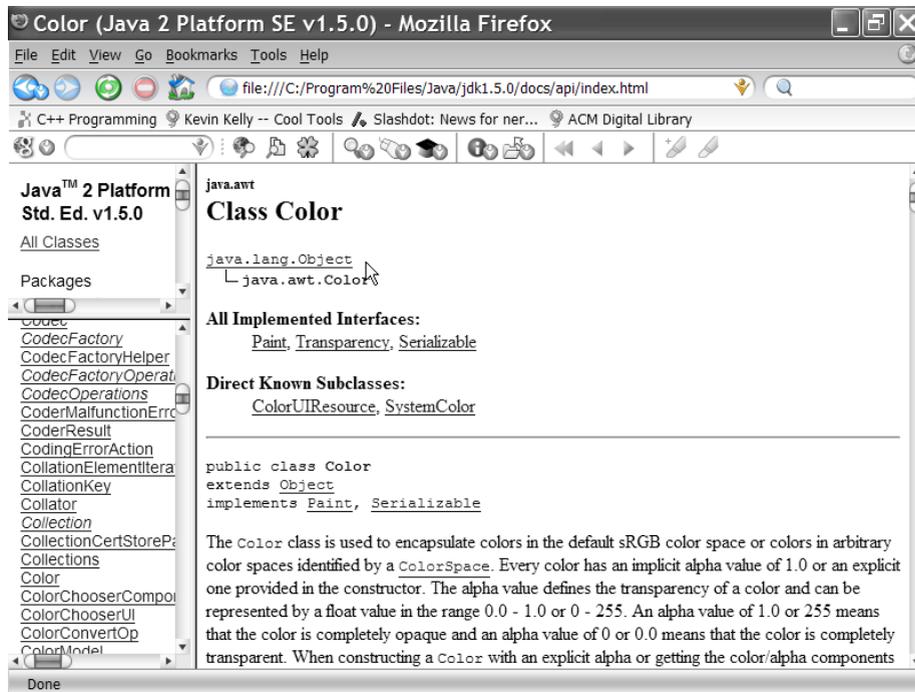


Figure 4.10: Top of the Color class.

4.5 Finishing EasyDice

Defining a Die

Just as we declared `EasyButton` above, we will declare the `OneDie` class here. We know the public interface of the class; all we need to do is provide definitions for the methods.

We want the die to display as a single digit on the screen so we will extend the `StringSprite` class. That gives us colors, scale, location, and all of that for free. We only need to keep track of the *state* of the die. What is the state of a die? Just the value showing on the die. That is an integer on the range [1-6] so we can have an `int` field.

```

1 import fang.core.Game;
2 import fang.sprites.StringSprite;
3
4 /**
5  * This class, extending StringSprite, represents one six-sided die.
6  * That is, a cube used for generating random numbers. The faces are
7  * marked from 1 to 6.
8  */
9 public class OneDie
10 extends StringSprite {
11     /** the value of the die; range is 1-6 */
12     private int face;
13
14     /**
15      * Initialize a OneDieStringSprite object with default values
16      */
17     public OneDie() {

```

```

18     setFace(1);
19 }
20
21 /**
22  * Get the current value of face.
23  *
24  * @return [1..6], the current value of the die
25  */
26 public int getFace() {
27     return face;
28 }
29
30 /**
31  * Roll this die: get currentGame(), use randomInt and setFace.
32  */
33 public void roll() {
34     setFace(Game.getCurrentGame().randomInt(1, 6));
35 }
36
37 /**
38  * Set value of face to newFace and update displayed value if newFace
39  * is legal; otherwise leave face unchanged.
40  *
41  * @param newFace the new value for face
42  */
43 public void setFace(int newFace) {
44     if ((1 <= newFace) && (newFace <= 6)) {
45         face = newFace;
46         setText(Integer.toString(face));
47     }
48 }
49 } // OneDieStringSprite

```

Listing 4.9: OneDie class

The face is a field because it is needed in all of the methods.

The constructor, lines 17-20 is simple: just set the face to 1. Using setFace to do the work here is an application of the DRY principle: if we set face and called setText, what if we changed how the number is displayed (say with pips?). Then we'd have to fix up the code in setFace, the constructor, and even, perhaps in roll. By having one method which does the setting and displaying of the state we have only one spot to have to update.

Lines 33-35 use getCurrentGame() as was done in EasyButton. We use it for randomInt to get a random integer on the range [1-6] and setFace to the new value. Again, calling setFace assures that all housekeeping for setting a face value is performed.

Because face is **private**, we must provide a way for classes using OneDie attributes²¹ to access the “value” of the die. We will add a *getter*; a method which provides read access to the value of a private attribute is referred to as a getter (it “gets” the value) and is typically named get + attribute name. Lines 26-28 return the current value of face.

A getter: (1) is **public** since it gives access to something not otherwise available; (2) returns the same type as the type of the attribute being “getted”; (3) typically does nothing more than return the value.

The one other thing we might want to do is set what face is up. This is particularly true for the two “point dice” in the upper-right corner of the design. They are not rolled but rather are set to a given value so the

²¹Notice that while we design OneDie we think of it as a component with attributes; when designing classes that use our new class, our class, itself, is a type for *attributes* of that class.

user can remember their point. As with a getter, a **private** attribute can have a *setter*. A setter takes a single parameter of the same type as the attribute and updates the value of the attribute. A setter, like `setFace` on lines 43-47, can protect the private field from being set to illegal values (like values outside [1-6] for a die). Line 44 protects the rest of the setting code from setting an illegal value.

Line 46 sets the `text` value of this sprite, updating the displayed value. The code `Integer.toString(face)` converts an integer to a `String` and a `String` is exactly the type needed to call `setText` on a `StringSprite`.

Given this definition, you might be asking yourself why we bother to declare `face` to be **private**. We needed 14 lines to define the getter and setter and, together, they pretty much provide the same access to the attribute as would have been provided by declaring the attribute to be **public**. Some trade off: fourteen extra lines or one fewer character.

Wait just a second, though. In the declaration comment for `face` it says the value should be limited to values 1 through 6. If `face` were **public**, then it would be possible to write the following code:

```
OneDie die;
...
die.face = 8;
```

Of course it is possible, with the current definition, to have the same effect (to violate the stated constraint on the type, that its `face` value be limited to integer values from 1 to 6) by writing:

```
OneDie die;
...
die.setFace(8);
```

But `setFace` only updates the value of `face` if it is valid. The logical “and” applied to the two comparisons will only evaluate to **true** when both subexpressions evaluate to **true**. The left subexpression is **true** only if `newFace` is greater than or equal to 1; the right subexpression is **true** only if `newFace` is less than or equal to 6. The two together are only **true** for the six values we consider in range.

Notice two things: (1) It was necessary to write two separate comparisons (the value being compared had to be typed twice); this is different than the way you might write it in mathematics but it is how it must be done in Java. (2) The way the two comparison expressions are written is done to indicate that the value of `newFace` is being constrained between 1 and 6; it is possible to write either comparison with `newFace` on either side of an appropriate comparison operator but this ordering was chosen to improve readability of the code.

At this point, having defined `EasyButton` and `OneDie`, the demonstration program we built using just their public interfaces should compile and run. Each time the button is pressed, the dice will roll; remember that sometimes they will randomly roll the same value. Also, make sure you press **Start** because the in-game button only functions when `advance` is called.

Now on to writing `EasyDice`. Note that from this point forward, we do not need to talk about `EasyButton` or `OneDie` except in terms of their public interfaces. This greatly simplifies our programming problem.

State of the Game

Consider the game described at the beginning of the chapter. Even given the two classes we have defined, the game is complex. What must we keep track of? The description describes a pile of matchsticks, a wager of a matchstick, a “point”, and four dice on the screen. Looking at what “point” means, we also need to be able to differentiate between a first roll and all subsequent rolls.

```
13  /** bank balance; game ends when this goes to 0 after a bet */
14  private int bank; // value
15  private StringSprite bankDisplay; // display
16
17  /** player's current bet */
18  private int bet; // value
19  private StringSprite betDisplay; // display
20
```

```

21  /** the button */
22  private EasyButton button;
23
24  /** the left die */
25  private OneDie dieLeft;
26
27  /** the right die */
28  private OneDie dieRight;
29
30  /** the value the player is trying to match */
31  private int point;
32
33  /** the left point die */
34  private OneDie pointDieLeft;
35
36  /** the right point die */
37  private OneDie pointDieRight;
38
39  /** is there an active bet? */
40  private boolean rolling;

```

Listing 4.10: EasyDice fields

This leads to a lot of fields. Lines 14-15 declare the bank (the pile of matchsticks) and lines 18-19 declare the wager. There are two variables for each because we track the number of matchsticks as an `int` and have a pretty display sprite with the description and value showing on the screen. It is always a good idea to label values you show to the user so they know what they mean.

The button, the four dice, and the point are all pretty much self-explanatory. The dice are named “left” and “right” to indicate which die of each pair they are and the dice in the upper left corner showing the current point have “point” in their name. Naming fields consistently greatly improves the readability of your code.

The last field, `rolling` is used to determine if we are rolling subsequent rolls in a round of easy dice (and trying to match the point before getting seven) or waiting to place a bet and make our first roll. There will be more on `rolling` when we define `advance`.

With the fields declared, what must `setup` do? Rather than try to list everything in detail, let's look at it from a high level:

```

setup button
setup bank
setup bet
setup dice

```

This looks like it is a top-level view in the process of stepwise refinement. In fact, it is, with each statement in the algorithm replaced with a call to a method.

```

64  public void setup() {
65      setBackground(getColor("green"));
66      rolling = false;
67      buttonSetup();
68      bankSetup();
69      betSetup();
70      diceSetup();
71  } // setup

```

Listing 4.11: EasyDice setup

The first two lines set the color of the game field and set the initial state to *not* rolling. The other four calls create, position, color, and add sprites to the game. In the interest of brevity, we will not be commenting on them in the text as they are very similar to the `setup` method in `RollDice`. Stepwise refinement starts at the top and works down until the described methods are easy to solve problems.

The `advance` method is really two methods. That is, there is work to do if we are waiting for the first roll and *different* work to do if we are rolling for a point. That sounds like an application of selection *and* stepwise refinement:

```

49 public void advance(double dT) {
50     if (!isGameOver()) {
51         if (!rolling) {
52             advanceWaiting(dT);
53         } else {
54             advanceRolling(dT);
55         }
56     }
57 } // advance

```

Listing 4.12: EasyDice advance

The real work for either case is pushed off until we define the given method. Each method, `advanceWaiting` and `advanceRolling`, sits and spins its wheels until the button is pressed. Thus the body of each is a big `if` (`button.isPressed()`) statement. If the button is pressed, the dice are rolled and the state of the game is updated accordingly.

```

98 private void advanceWaiting(double secondsSinceLastCall) {
99     if (button.isPressed()) {
100         // place and show wager
101         bet = 1;
102         bank = bank - bet;
103         betDisplay.setText("Bet: " + bet);
104         bankDisplay.setText("Bank: " + bank);
105
106         int roll = rollTheDice();
107
108         // check for a win on the first roll
109         if ((roll == 7) || (roll == 11)) {
110             win();
111         } else {
112             // copy roll dice up to the point
113             pointDieLeft.setFace(dieLeft.getFace());
114             pointDieRight.setFace(dieRight.getFace());
115
116             // set new point, set rolling flag, and change button text
117             point = roll;
118             rolling = true;
119             button.setText("Roll for point");
120         }
121     }
122 } // advanceWaiting

```

Listing 4.13: EasyDice advanceWaiting

In `advanceWaiting`, when the button is pressed a wager is made. Then the dice are rolled. If the roll is a winning number for a first roll, call the `win` method. Otherwise update the point dice to display the same faces as the game dice (lines 113-114), point to be the sum of the dice (line 117), set `rolling` to `true` since we are now rolling for a point, and finally, change the button text to indicate that we're rolling for point.

```

79 private void advanceRolling(double dT) {
80     if (button.isPressed()) {
81         int roll = rollTheDice();
82         // game wins, loses, or keeps going. Nothing to do to keep going
83         if (roll == 7) { // lose
84             lose();
85         } else if (roll == point) { // win
86             win();
87         }
88     }
89 } // advanceRolling

```

Listing 4.14: EasyDice advanceRolling

In `advanceRolling`, when the button is pressed we roll the dice. If roll equals 7 the player loses. Else, if the roll equals the point, the player wins. In any other case, rolling continues. Note that `win` and `lose` are defined to update the wager, the bank, and to set `rolling` **false**.

4.6 Summary

An expression is a piece of code which returns a value. Simple expressions are literal values, variables, **new** expressions, and method calls. Simple expressions have types. Expressions are built up of simpler subexpressions by combining the subexpressions with *operators*

Literal values, like 3.1415 or "One Two Three" are evaluated by the compiler and compiled directly into a program. Variables, fields and local variables alike, are names for values which are filled in (using assignment) at run-time. The **new** operator allocates memory and calls a *constructor* to initialize the space into an object. Method calls, where a method is named and parameters (if any) are provided, can return values.

Operators have *precedence* which determines the order in which they are applied. Thus $4 + 3 * 2$ is 10 ($4 + (3 * 2)$) rather than 14 ($((4 + 3) * 2)$). Parentheses can be used to specify order of application and to make it clearer even when they are not required.

All expressions have types. The **extends** relationship between two types means that the child class *is-a* or is an example of its parent class. A every running shoe is a shoe, after all.

Operators can be combined with assignment to simplify typing (and to clarify intent for a reader familiar with the Java language). For example

```
vowelCounter += yCounter;
```

This adds the value of the `yCounter` variable to the current value of the `vowelCounter` variable.

Since almost any Java class can be extended into another class, an important design decision is what class to extend. In this chapter we extended a `Sprite` with additional state to play our game. More guidance on this in the next two chapters.

FANG Classes and Methods

Java Templates

```
<returnStmt> := return <expression>;
```

```
<returnStmt> := return;
```

Chapter Review Exercises

Review Exercise 4.1 State which of the following collection of possible identifiers are *not* legal in Java and why they are not legal:

- (a) oneFish
- (b) ReD
- (c) findSmallest
- (d) Bilbo Baggins
- (e) NDX
- (f) ingredient
- (g) 2fish
- (h) Go!
- (i) FarFetched
- (j) 123BrockportLn
- (k) mouseEvent
- (l) FryingPan
- (m) SIZE_OF_SQUARE
- (n) Alan_Turing
- (o) flash cards
- (p) student-grade-point-average
- (q) LionTigerLeopardPumaCougarCount
- (r) Orca#
- (s) alpha_Bravo
- (t) BrockportLn123

Review Exercise 4.2 Classify each of the following identifiers as either a type (**class**) name, a method, a variable, or a constant.

- (a) DirtyDozen
- (b) color
- (c) createCheckerKing
- (d) YEAR_LINCOLN_WAS_BORN
- (e) myGameProgram
- (f) GolfClub
- (g) findSmallest
- (h) LENGTH_OF_SHIP
- (i) nameOfCountry
- (j) getCaptainsName
- (k) MAXIMUM_NUMBER_OF_PIRATES
- (l) PlayingCard
- (m) currentPokerHand
- (n) getHighCard
- (o) ChessPiece
- (p) getRaspberryCount
- (q) firstMove
- (r) castleToKingside
- (s) numberOfRaspberries
- (t) SmallInteger

Review Exercise 4.3 What is the scope of the variable `blueLine` in the following listing:

```

class Something
  extends Game {
  public void gamma() {
    (a)
  }

  public int delta(int a) {
    (b)
  }

  LineSprite blueLine;

  public void setup() {
    (c)
  }
}

```

Review Exercise 4.4 What is the text in the label after this code is executed:

```

StringSprite results = new StringSprite();
addSprite(results);

int x, y, z;
String cubit = "atom mixed replication";

x = 10;
y = 100;
z = y;

y = 2 * y;

x = z * x;

results.setText(cubit.substring(1,7) + " (" + x + ", " + y + ", " + z + ")");

```

Review Exercise 4.5 What color is the RectangleSprite displayed on the screen filled with?

```

RectangleSprite lemon = new RectangleSprite(0.2, 0.2);
lemon.setColor(getColor("yellow"));

RectangleSprite lime = lemon;
lime.setColor(getColor("green"));

addSprite(lemon);

```

Programming Problems

Programming Problem 4.1

Take the Snowman.java program as a base and design a CompositeSprite-extending class, SnowmanSprite. This can be done in two steps.

- (a) Make a `SnowmanSprite.java` file and build a constructor for the class. The constructor should instantiate three `OvalSprites` of appropriate sizes and position them so that the center of the snowman is the center of the middle ball *and* the overall height of the snowman is 1.0 screens (width is determined by the width of the bottom ball).
You can test this sprite by building a simple `Game` which adds ten sprites with random locations and random rotations. You might want to try setting random colors for each as well.
- (b) Modify `SnowmanSprite` by overloading the `setColor` method. While `CompositeSprite` has a `setColor` method, the method does not set the color value of sprites within the composite.

Programming Problem 4.2

Start with `EasyDice.java`. Modify the program for two players alternating play on the same machine.

- (a) What variables will need to be duplicated? Which will not?
- (b) How will you keep track of which player's turn it is? When will which player's turn it is change? You may assume that the players are able to pass the mouse from one to the other when the turn changes.
- (c) When is the game actually over? What message could you show?

Programming Problem 4.3

Pig is a simple dice game for children. Two players begin with a score of 0, adding some number from rolling a single die on their turn to their score. The first to 100 wins the game.

On their turn a player rolls one die at a time and keeps a running total of the rolls. At any time the player may end their turn and add the total for all rolls that turn to their total. A player's turn is also over when they roll a 1 or a "pig"; when a player rolls a pig their turn ends and they receive 0 points for the turn.

Design a two player (alternating on the same machine) game of *Pig*. You should display both scores, whose turn it is, the turn score, and two buttons (one for ending the turn, one for rolling again). The game logic becomes a bit more difficult in tracking two different buttons.

Rules: Methods, Parameters, and Design

We have seen how to use top-down design to go from a description of a of public interfaces. This chapter continues that while going deeper into the *delegation* problem solving technique through the use of *methods*. We will also create an animating countdown timer using *iteration* through FANG’s built-in iteration of the advance method.

5.1 A Simple Arcade Game: SoloPong

One of the earliest commercially-successful computer games was Atari’s Pong coin-operated video game. Two players, two paddles, and a simple game of video ping pong. With the venerable original in mind, we will build a FANG version that is similar to a solo handball game. The ball will start at a random location on the left end of the screen and the bounce off of the walls; to give the player a warning, whenever the ball is being started, a 3-second countdown will precede the release of the ball.

The *SoloPong* is diagrammed in Figure 5.1. The ball will move on its own (just as the apple and cat did in previous games); the ball will also bounce off of three edges of the screen. The paddle will move in response to the up and down arrow keys. These and the countdown timer will all be sprites with their own state (when you see “*something with state*” you should be thinking about creating a new class extending the *something*).

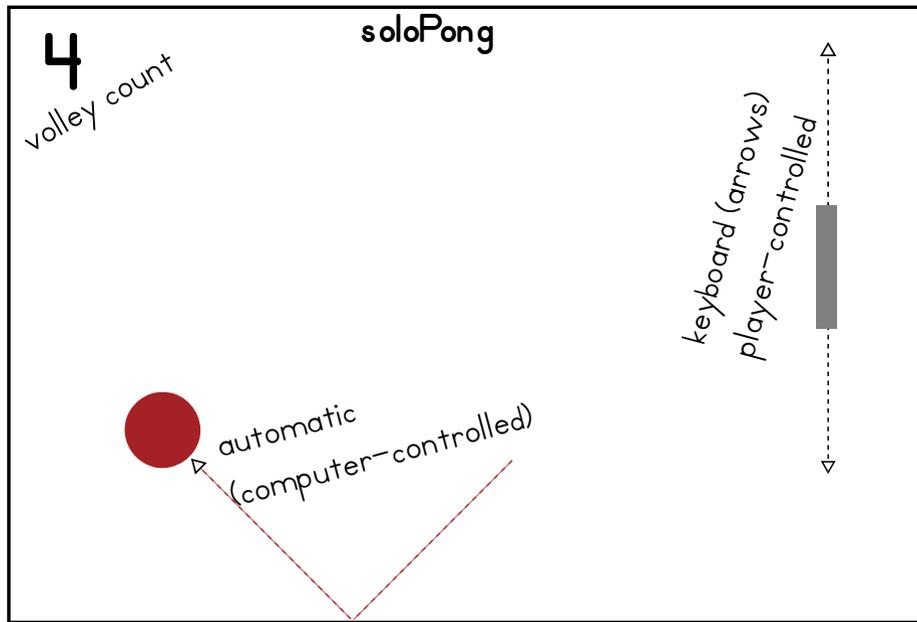
Physics of Bouncing

What makes a ball different from an `OvalSprite`? What made an apple different from an `OvalSprite`? The fact that we moved it in advance. Given that we are putting animation code into our game objects, it seems that a ball differs from an `OvalSprite` because it has a velocity. By tracking its own velocity, a ball can be told to move simply by giving it the elapsed time. Further, if a ball tracks its own velocity, it can update that velocity to bounce off of walls and other sprites. Even better, if *each* ball tracks its own velocity, each can have a different velocity meaning each can bounce off of walls, each other, etc. independently.

So, how can you keep track of velocity. Our games are two-dimensional so velocity is a distance to move in a given time in two dimensions. This could be stored as a *polar coordinate* or a heading and a magnitude (which direction to move and how far to move in a unit time) or as *Cartesian coordinates*, an x-component and a y-component of velocity. Either method will work but if most of our bouncing is off of vertical and horizontal “surfaces”, it is very easy to simulate that with Cartesian coordinates.

What we are doing, simulating motion and collision and bouncing, is simulating *physics*. We are creating rules, more complex than those in `NewtonsApple`, which create a believable simulation of the real world¹. Since

¹for a sufficiently loose interpretation of believable

Figure 5.1: *SoloPong* game design diagram

we are simulating physics, we will call the two components of the velocity in our `PongBall` `deltaX` and `deltaY`; this is because in physics Δ , the Greek letter delta, is used to mean change so Δ_x and Δ_y are the change in position in the x-coordinate and y-coordinate.

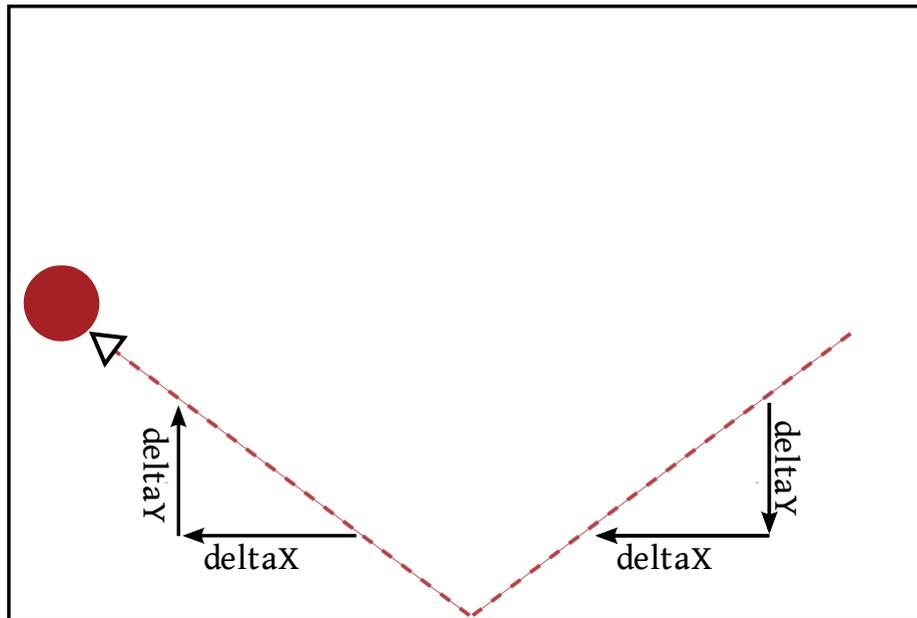


Figure 5.2: A bouncing ball

How does a ball bounce off of the floor? In Figure 5.2 is a ball moving from right to left and striking the bottom edge of the screen, the floor. Notice that the angle of incidence equals the angle of reflection (or: the

angle between the floor and the movement line of the ball is the same on either side of the point of reflection). The component velocities, Δx and Δy are drawn to show how they combine to create a diagonal path for the ball; they also show how we can simulate the physics of a bouncing ball.

When a ball hits a surface, the component of the velocity perpendicular to the surface struck reverses direction. The new velocity has the same size or *magnitude*, just the direction has changed². In FANG terms, if the ball strikes a horizontal surface, the y-component of the velocity (perpendicular to the surface struck) changes sign. The x-component (parallel to the surface struck) is unchanged.

Reacting to a collision with *any* surface takes the same form; the portion of the velocity perpendicular, or *normal*, to the surface struck is reversed in direction. The two hard parts of collision reaction are determining *where* two objects struck one another and *what* the normal of the surface at the point of collision is.

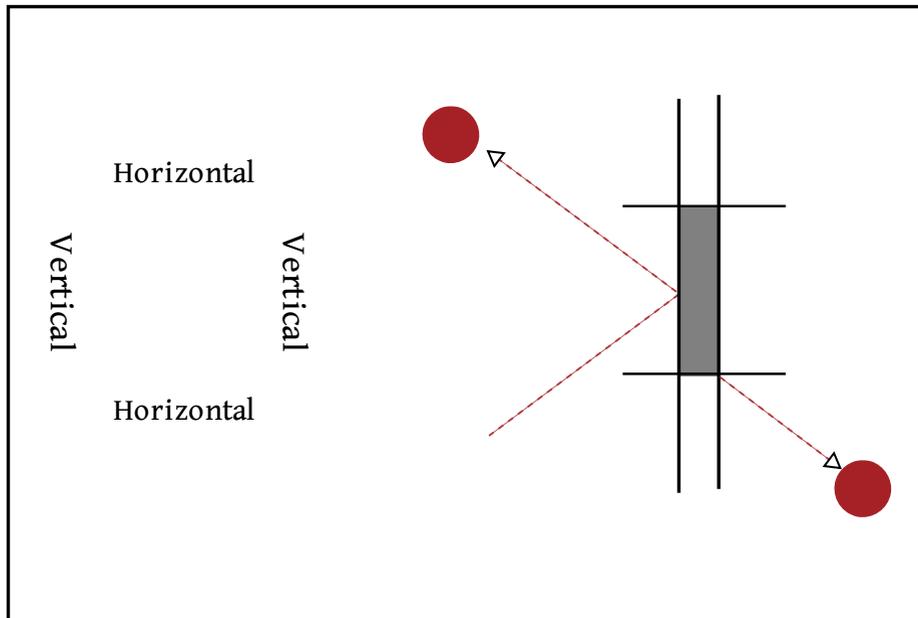


Figure 5.3: Bouncing and bounding boxes

A first approximation of an arbitrarily shaped sprite is to use its *bounding box* or the smallest axis-aligned rectangle that contains all of the sprite. Using a bounding box means that we only reflect off of horizontal or vertical surfaces. Figure 5.3 shows how a bounding box would fit around a grey rectangle sprite. The two balls bouncing off of it show how a ball would bounce if it hit one edge of the bounding box or if it hit two edges at the same time. To the left, the four regions are marked with slanted lines to show where the ball struck a vertical or a horizontal edge. In the corners, the moving ball bounces off of the corner or off of both a horizontal and a vertical surface as shown by the cross hatching in the diagram.

Objects with Their Own Advance

What sort of objects are the ball, countdown timer, and paddle that are part of *SoLoPong*? That is, what type of sprite should each extend and what extra information does each need?

What state must the ball have? We want to know where the ball is (its x-coordinate and y-coordinate): that is already part of *Object*. We also need to know how to *move* the ball each advance: that means we need to know the x-velocity and the y-velocity of the ball.

²This assumes the collision is completely elastic or that a ball bounces to the height from which it was dropped. We will later see how to simulate a “real” bouncing ball.

The countdown timer is like `OneDie`: it is a `StringSprite` extended to track a number (in this case a number of seconds). Each advance it will modify the timer and it will change the color of the display, fading the displayed value into the background.

The paddle must keep track of its position (something `RectangleSprite` already does). During each advance it must check whether one of the vertical arrow keys is pressed; if so the paddle will move in the correct direction. It would also be good if the paddle remained on the screen regardless of arrow keys being pressed.

All games we have studied so far handle their animation directly from the `Game`-derived class. Most use `advance` directly while `EasyDice` used two different “advance” methods, one for each major state the game could be in.

For the sake of argument, we will sketch all of the responsibilities of `advance` would have if we used this approach for `SoloPong`:

All games so far handle all animation directly in the `advance` method of `Game`-extending class. Let’s sketch what that one method would have to handle *if* we used that approach. This game is much more complex than any we have yet written.

```

if (waiting to start play)
  if (spacebar pressed)
    start countdown timer
if (countdownTimer is counting down)
  decrement timeRemaining
  if (timeRemaining <= 0.0)
    countdownTimer NOT counting down
  else
    animate based on time remaining
else
  move ball according to ball velocity
  move paddle according to keyboard
  if (ball hit wall)
    if (wall is bounce wall)
      update ball velocity
    if (wall is score wall)
      update score
      move ball to start position
      wait to start play
  if (ball hit paddle)
    update ball velocity

```

This is quite complicated. As has been mentioned, one way computer scientists control complexity is to decompose the problem into smaller, simpler problems. The solutions to the simpler problems can then be composed together to solve the original problem. This `advance` method would benefit greatly from this approach.

What if the ball, paddle, and countdown timer each had their own `advance` method. Then the `Game`’s `advance` method is greatly simplified:

```

if (waiting to start)
  if (spacebar pressed)
    countdownTimer.startTimer(3)
  if (!countdownTimer.isExpired())
    countdownTimer.advance(deltaT)
else
  ball.advance(deltaT)
  paddle.advance(deltaT)

  if (ball.isOutOfBounds())
    wait to start

```

Why call the parameter of `advance deltaT`? For two reasons: to demonstrate that we can call parameters anything we want to, anything that makes the code easier for the human reader to understand. Δ_t is the change in the variable t or time. We will use the whole word `deltaT` at the moment though we might shorten it in future chapters.

5.2 Top-down Design

Top-down design is decomposing a complex problem into a collection of less complex problems along with a solution to the complex problem composed of solutions to the simpler problems. This is how we designed the `OneDie` and `EasyButton` classes. We went from a public interface down to running implementations. We will review the process here and take a deeper look at the different levels of abstraction.

Pretend that it Works

The first step of stepwise refinement is a game of “let’s pretend”. When working at a higher-level, decompose the high-level solution into a composition of solutions of lower-level, simpler problems. Define the information necessary to solve the simpler problems and figure out how the high-level solution can provide that information to the lower-level solutions. This means examining the parameters necessary for the higher-level solution and determining how they are passed on to the lower-level solutions.

So, how would we write the actual `advance` method for `SoloPong` using the “pretend it works” approach? We will assume that we have working `ball`, `paddle`, and `countdown timer` implementations. We will also assume that fields with appropriate names have been defined for them. Then we could try something like:

```

164     if (getKeyPressed() == ' ') {
165         beginCountingDown();
166     }
167 } else if (isCountingDown()) {
168     countdown.advance(deltaT);
169     if (!isCountingDown()) {
170         beginPlaying();
171     }
172 } else if (isPlaying()) {
173     paddle.advance(deltaT);
174     ball.advance(deltaT, paddle);
175
176     if (ball.isOutOfBounds()) {
177         beginWaitingToCountdown();
178     }
179 }
180 }
181 } // SoloPong

```

Listing 5.1: `PongBall`: `advance`

There are three states the game can be in: waiting to start, counting down to start, or playing. Since we are designing from the top down, it is not a good idea to worry too much about how, specifically, we will know which state we are in. We can just pretend that we can write three Boolean methods: `isWaitingToCountdown`, `isCountingDown`, and `isPlaying`. These methods hide the details and let us delay making a decision about how they will work.

The problem with putting off the decision is that we don’t know how to change the state of the game. That is, when the countdown timer finishes, we want to start playing. If we don’t know how `isPlaying` works, how can we start the playing? We have to pretend that a method called `beginPlaying` exists that will do whatever must be done to change to playing mode. Similarly `beginWaitingToCountdown` and `beginCountingDown` will shift the game into each of those states.

The important thing to note here is that we have pushed the details down to another level of abstraction and we can write the whole game control right here, right now.

Refining Rules: Multiple Levels of Abstraction

The public interface of a class provides the programmer with a description of the services the class. The programmer who reads the documentation of the public interface should be able to design programs *using* the class without any additional information on how the class is implemented. We will now look at one of the lower levels to which we pushed off our problems, the countdown timer.

Determining Necessary Methods

A digital kitchen timer, the kind used to make sure oatmeal cookies don't burn, serves as an analogy for our countdown timer class. The timer permits you to set the countdown value, to start the countdown, to see remaining time, and to see if there is any remaining time (has the countdown expired?). If we were to model such a timer as a Java class, these capabilities would translate to **public** methods:

```
class CountdownTimer {
    public void setTimer...
    public void startCountdown...
    public double getRemainingTime...
    public boolean isCountdownExpired...
}
```

Is there anything missing (from the public interface)? While in the kitchen we would go into the junk drawer to find the kitchen timer, we probably want to be able to create a timer in Java: that implies a public constructor (most classes have public constructors). We also need to be able to supply the countdown timer with a "tick". Where as a physical kitchen timer has a clock circuit built into it, we are going to have to tick the countdown timer as part of a Game; that implies the timer needs its own advance method (the method we know which has elapsed time as a parameter, at least in Game; we will copy the signature):

```
class CountdownTimer {
    public CountdownTimer...
    public void setTimer...
    public void startCountdown...
    public double getRemainingTime...
    public boolean isCountdownExpired...
    public void advance(double deltaT) { ... }
}
```

Signatures

What do the remainder of the signatures look like? When setting the time, we need to know what time to set it to. Thus there should be a parameter indicating the remaining time to set. The other four methods don't seem to need any parameters (though it might be a valid design to have a constructor which took the initial countdown time; we will separate the construction and setting in our class). Also, if we intend to have the timer appear on the screen, displaying the countdown, we should extend a sprite class capable of displaying a string. Thus the class's public interface looks like this:

```
class CountdownTimer
    extends StringSprite {
    public CountdownTimer() { ... }
    public void setTimer(double remainingTime) { ... }
    public void startCountdown() { ... }
```

```

public double getRemainingTime() { ... }
public boolean isCountdownExpired() { ... }
public void advance(double deltaT) { ... }
}

```

A design rule of thumb is to give all methods and fields as restrictive an access level as possible. The reasoning is that the fewer places a method can be called or a field be manipulated, the easier it is to find mistakes involving that method or field. We will continue to declare all fields to be **private** and will begin, with this chapter, to try to define only those methods which form the public *interface* of the class as **public**. There might be other, implementation methods that are not specified here.

We use the public interface of `CountdownTimer` (and the public interface of `StringSprite` because we got that for free by extending that class). It is not necessary for the user to see what happens inside the class. `JavaDoc` documents are generated automatically by determining the public interface of a class and extracting the comments from those methods.

Thus the public methods combine to advertise what the object they belong to is capable of. They do not advertise just how it does it but they provide an indication of what the user must do in order to use the object effectively. The `advance` method should make sure that it includes comments that it must be called from the `advance` method of a `Game` with the same `deltaT` in order for the countdown to work.

5.3 Delegation: Methods

A *method* is a named sub-program. As a sub-program, it is possible to think of it as a complete program, just one that solves a smaller problem than playing an entire game. Thus all types of statements and ways of combining statements are available in any given method definition.

The other thing to keep in mind is that when writing a method, *defining* a method, Java, in fact, does not do anything except record a newly named command. You can think of it as a recipe: by writing a recipe in a cookbook, the author has defined *how* to make delicious oatmeal cookies; the definition, however, produces no cookies. Instead, when some cook applies the appropriate ingredients (parameters) to the recipe (method), when they use the *how*, they produce warm, wonderful cookies. Or whatever the recipe defined, in any case.

Defining New Rules: Declaring Methods

A method must be defined inside of a class definition. The first line of a method definition is known as the *signature*. As mentioned previously, a method signature has four parts: the access level, the return type, the name of the method, and the parameter list.

The access level is either empty or one of three keywords: **public**, **protected**, or **private**. Defining the extremes first, **private** methods (and fields; these same access levels apply to field definitions, too) can only be called from other methods defined in the exact class definition containing the **private** method. A **public** method can be called from methods in any class at all³.

A **protected** method behaves as a **private** method for all classes *except* classes extending (directly or indirectly) the class where the method is defined; for child classes, **protected** methods are visible and can be called.

The empty access level (none of the three keywords) is also a mix of **public** and **private**: all classes defined in the same folder treat it as **public**, all other classes (even ones that extend the original class) treat it as **private**.

The return type is just a type, as when declaring fields and variables, or the keyword **void**. Calling a **void** method is not an expression, it is a statement because a **void** method cannot return a value (which is the definition of what an expression does).

The name of the method is any valid Java identifier (starts with letter or underscore, continues with letters, underscores, and digits). Naming conventions, or how to select *good* method names was discussed in the previous chapter.

³Yes, a **public** field could also be seen and modified from any class, not just the one with the field definition. This is terrible design practice and we will not use public fields in this book.

Finally, the parameter list is a list of zero or more type and name pairs (the meaning is investigated in Section 5.3 below). Parameters communicate values from the caller to the called method.

Calling a Method

When rules are combined sequentially, execution begins at the beginning and follows each rule in turn until the end. With selection, some condition is evaluated and if it is true one set of rules is followed and if it is false a different set is followed⁴. How does delegation work?

Delegation has two parts. The first is defining the method. The signature and the body of the method together define a newly named rule. Note that the definition of the rule does not, by itself, do any computation when the program containing it is run. The method only acts when it is *called*.

Consider living on a college campus. Over time you will internalize where you are living. You will be able to find your way home from the bookstore, the dining hall, and even your computer science lab. That internalized knowledge of *how* to get back to the room, the directions you know, is a definition of a “go home” method in your head. Having it defined in your head does not do anything. Something happens only when you tell yourself: “Now, go home from here.”

When you call your method, your internal directions evaluate the parameter, “here”, and you are off. It is possible that going home was part of some larger sequence of instructions (“change clothes for volleyball”, for example). The key is that “go home” does not know anything about that. In fact, the larger set of instructions is suspended until you complete “go home” (which is a good thing if the next step in changing for the game is “remove pants”).

Analogously, when `setup` calls `someSprite.setColor(blue)`, the parameters are evaluated and then the instructions in `setup` are suspended. The method stops making progress while `setColor` executes. The rule follower which was performing the instructions in `setup` sets a bookmark so it can return to the exact spot where that method was suspended and then execution continues with the first line of the body of `setColor`. By default, the body is evaluated sequentially; using `if` statements, iteration, or further delegation can change the default behavior. When the rule follower finds a `return` statement or run off the end of the body of `setColor`, then it returns its attention to the bookmarked spot in `setup`.

The definition of a method constructs a new, named rule. Calling the method suspends the method containing the call, marking the *return address* (the bookmark) to which control will return and it executes the definition from the beginning. It is possible to have more than one copy of a given method running at the same time though at most one of them is the current, active method.

Parameters: Customizing Rules

Returning to the cooking analogy for a moment, consider a step in the cookie recipe which says Cream together the butter and the sugar. What does that mean? Fortunately, in a fairly complete cookbook such as *The Joy of Cooking*, there will be a section on techniques and the novice cook can look up the definition: Mix the ingredients together until they are light and creamy.

What are the ingredients? That is, in the definition of the technique, there is discussion of something which the writer of the definition does not yet know. It is something that cannot *be* known until the technique is used in a recipe; until the technique is called.

In the cookie recipe, the call is for creaming together the butter and the sugar; that is the value for which the ingredients serves as a placeholder. When we apply the technique at this point in the cookie recipe, those are the actual ingredients used in the mixing.

Consider a liver dumpling recipe. It might contain the instruction Cream together the egg, liver paste, and breadcrumbs. What are the ingredients this time? It is the list provided when the technique was called.

The analogy is good but only goes so far: in Java, the placeholders are local variables, declared between the parentheses in the signature line of the method declaration; there must be one parameter in the signature for each parameter in the call and the types must match⁵.

⁴If there is no `else` clause on the `if` statement, then the alternate, `false` path is just an empty list of rules.

⁵This is the biggest problem with the analogy; in the cookbook, the ingredients applies to any number of items specified in the call. In Java you must specify exactly the number of items when defining the method.

While it seems limiting to have to know exactly what types of parameters are necessary when defining a method, it can actually be quite nice: it is possible to define different methods with the same names so long as the parameter lists are different.

For example, we have used multiple versions of `getColor` from `Game`, including ones with the following signatures:

```
public Color getColor(String colorName)
public Color getColor(String colorName, int opacity)
public Color getColor(Color currentColor, int opacity)
```

Java can tell them apart by the list of parameters we provide when we call it. This brings up a point we have overlooked until now: rather than one *parameter list*, there are really two different things that are parameter lists: the parameter list which is part of the signature of the method and the list of parameters included when the method is called. These are known as the *formal* parameter list and the *actual* parameter list respectively.

The formal parameter list, as has been said before, is a list of zero or more `<typeName><parameterName>` pairs separated by commas. By appearing in the parameter list, the `<parameterName>` is just like a *local* variable, local to the body of the method for which it appears as a formal parameter. The type of the parameter determines the available public interface and it can be used as any other local variable. The difference is that the initial value of the parameter is not set in the definition of the method but rather when the method is called.

The actual parameter list is the list of expressions between parentheses right after a method is called. The `getColor` examples above are signatures of methods, each with a formal parameter list; below is a list of calls to those methods (assume in some method defined in a `Game`-extending class):

```
Color dummyColor;
dummyColor = getColor("navy");
dummyColor = getColor("misty rose", 64);
Color otherColor = getColor(dummyColor, 255);
```

The values in the actual parameter list are assigned to each of the parameters in the formal parameter list in the order they appear from left to right. That is, in the second call to `getColor`, `colorName` is initialized with the `String` value `"misty rose"` and `opacity` is initialized with the value `64`.

Passed by Value

The important thing to note is that all Java parameters are passed by value. This means that each actual parameter expression is evaluated before the method is called and the result is assigned (copied) into the corresponding formal parameter.

This means that all changes made to the formal parameter inside the body of the method happen to the *copy* of the value. When the called method returns control to the caller, the parameter goes away. Any changes made to the parameter go away with it.

First we will consider what happens when we pass plain-old data into a method. Assume the following is part of a `Game`-derived program.

```
public void fiveTimes(double x) {
    x = 5 * x;
}

public void init() {
    StringSprite s = new StringSprite();
    double k = 9.0;
    fiveTimes(k);

    s.setText(Double.toString(k));
    addText(s);
}
```

Assuming no other changes are made to the text in `s` and `s` is added to the game, what value does `s` display on the screen? Let's trace the execution of `init` using what we know about delegation.

The `StringSprite s` is declared and constructed. Then the `double k` is declared and the value 9.0 is stored in it. Remember, plain-old data types behave like boxes into which values are stored (rather than as references). Then a call is made to `fiveTimes`.

What happens when a method is called? The parameter expressions in the actual parameter list are evaluated. `k` is a variable so it evaluates to its current value or 9.0. Then the value of the expression is used to initialize the formal parameter in the corresponding position. First actual parameter evaluates to 9.0 which is used to initialize the value of the first formal parameter, `x`. Note that initialization means initial assignment so the value is *assigned* to `x`.

Then execution of `init` is suspended (with a bookmark indicating to continue just after the call to `fiveTimes` (which is the semicolon at the end of the line so the line is finished and execution continues below that) and execution of the body of `fiveTimes` begins.

The one line in the body of `fiveTimes` is an assignment statement. We evaluate the right-hand side of the assignment and then assign that value to the variable on the left-hand side of the assignment operator. The expression, `5 * x`, is evaluated by getting the value of `x`, coercing the `int 5` to a `double`, 5.0, and then multiplying them together. That is, `5.0 * 9.0` is evaluated giving 45.0.

The value is assigned to `x`. That means that in `x` the 9.0 is clobbered with the new value, 45.0. Execution runs off the end of the body of `fiveTimes` so control returns to the bookmark. Execution of `init` continues from where it was suspended. Thus the value of `k` is passed as a parameter to the `Double.toString` method to convert it to a string representation of the same number. If `k` contains 45.0, `Double.toString` returns "45.0" and if `k` contains 9.0 it returns "9.0". What value does `s` display on the screen?

Plain-old data is stored in a memory location. When a copy was made from `k` to `x`, both `k` and `x` contained the value 9.0 but they were separate copies of that value. Thus when `x` was changed, `k` remained unchanged. Thus, with pass by value parameters, it is not possible for changes within a method to directly modify any plain old data parameters.

What about object parameters? What if we passed in a `StringSprite`? Could you change the value. Fundamentally the answer is, "No." Yet object variables are not boxes like plain old data type variables; object variables are references or labels. That makes things different. Consider the following modified version of the above code:

```
public void fiveTimesString(StringSprite p) {
    String content = p.getText();
    String newContent = content + content + content +
        content + content;
    p.setText(newContent);
}

public void init() {
    StringSprite s = new StringSprite();
    s.setText("q");
    addText(s);

    fiveTimesString(s);
}
```

What value does the `StringSprite s` display? The first three lines of `init` are straight forward: declare a sprite variable, construct the sprite, set the text of the `StringSprite`, and add the sprite to the game. The last line is just a call to a method with `s` as the actual parameter.

The expression, `s` is evaluated and the result, a reference to the given `StringSprite`, is used to initialize the corresponding formal parameter. This is equivalent to `p = s` where `p` is the formal parameter variable name and `s` is the actual parameter expression.

Then `init` is suspended, execution runs the statements in the body of `fiveTimesString`. The first line labels the text in the `StringSprite` as `content`. Then the `String` concatenation operator is used to paste five copies

of the content together and label the result `newContent`. Then `p.setText` is called and the method finishes. `init` resumes execution just after the call to `fiveTimesString` was called and `init` finishes, returning control to the method which called `init`.

The question is, what does the `StringSprite` on the screen show?

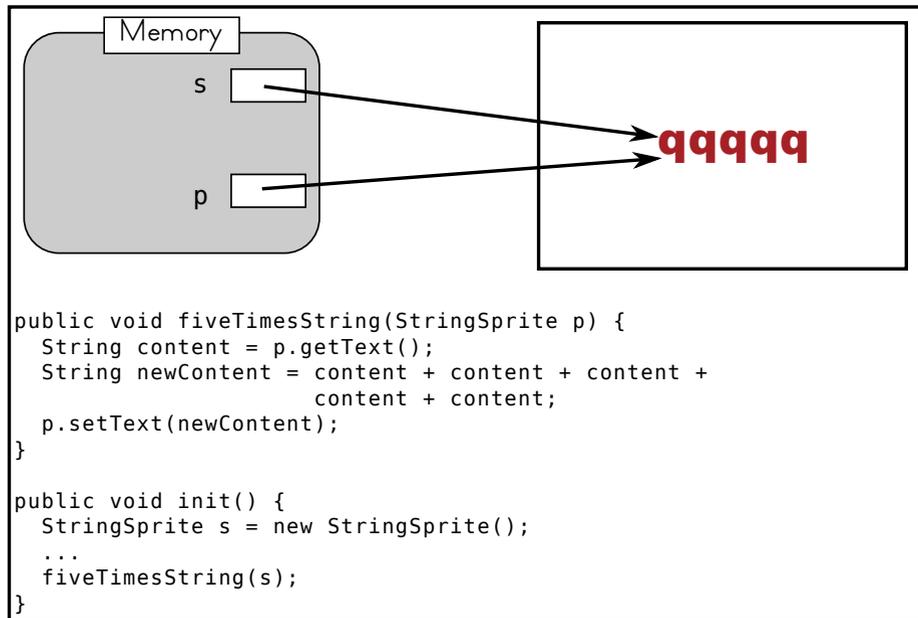


Figure 5.4: Java Object Parameters

Drawing a picture of what goes on in memory can be helpful here. When `s` is assigned a value from `new`, the box called `s` refers to the `StringSprite`. When `p` is initialized from `s`, the contents of box `s` are copied into `p`. Note that in the picture, `p` and `s` refer to the *same* sprite. This is the same situation we had in Figure 4.5 when we assigned one object variable to another. In this case, the `StringSprite` is set to the text "qqqqq".

This is somewhat confusing for beginning programmers: if the parameter is passed by value, how can the sprite be changed. The distinction is between the content of `s` which cannot be changed because it was copied when the method was called and the thing referred to by `s`, an object which is shared through the references of both `p` and `s`. Consider what would happen if `fiveTimesString` assigned the results of `new` to its formal parameter. That is, add as a second line in the method `p = new StringSprite();` and leave the rest unchanged. What happens.

`s` and `p` are different boxes in memory; changing `p` does not change `s`. After the assignment, `s` and `p` also refer to two different objects (the results of two different calls to `new`). Thus the changes to the newly minted `StringSprite` have no impact on the old `StringSprite` so the value on the screen in this case remains "q".

Methods as Functions: Returning Values

Methods can return values. That is why the signature includes a return type. When the return type is `void`, the method does not return any value. When the return type is something else,

```

64 }
65
66 /**
67  * Has the count expired?
68  */
69 }
70
71
72
73 }
74

```

75 `/**`

Listing 5.2: CountdownTimer: functions

The two methods shown above are the two functions in the `CountdownTimer` class. The first, `getRemainingTime` is what we have been calling a *getter*, a method for getting the value of a `private` field. The `return` statement takes an expression (here a single variable) of the right type, evaluates it, and sets the return value of the method to that value. Thus, if `remainingTime` were 10.34 and I wrote the following in my Game-derived class:

```
double d = countdown.getRemainingTime();
```

Then `d` would be set to 10.34.

The other function here is `isExpired`. Java naming conventions suggest that `boolean` methods be named `is<Something>` where `<Something>` describes what it means when the method returns `true`. So, what does `countdown.isExpired()` going to return (the listing above has cut off the header comment for the method)? Looks like it will return `true` if the timer has expired. Thus `isExpired` lets an outsider see that the alarm is ringing and we should do something.

Delegation is the creation of named units of computation. It works very well with stepwise refinement because each lower level can be defined in terms of methods to implement the solution of the various sub-problems. Methods can use parameters to specialize how they work.

5.4 Expressions Redux

The following snippet of code ends with an *expression*. An expression is a piece of code which returns a value. A value has both a *type* and a *value*. What are the type and value of the expression?

```
int p = 1001;

"p = " + p
```

The expression is an application of the `+` operator. The type of an addition expression depends on the types of the subexpressions being “added”. If they are both `int...`, no, not the case here. If one is an `int` and the other is a `double...`, no, not that case, either. What is the type of `"p = "`? It is a sequence of characters enclosed in double quotes. It is a `String`⁶ literal. A `String` followed by a plus sign followed by a variable name.

Java uses `+` for *concatenation* of strings; when the right-hand value is a variable, the variable is converted to a `String` first and then tacked onto the end of the left-hand side. So, in this case, `p` has the value 1001 so it is converted to the `String` `"1001"` and the result is the `String` `"p = 1001"`.

When concatenating a `String` with any plain old data type, Java uses the `<Type>.toString(<type>)` method where `<Type>` is the name of the type with a capital letter and `<type>` is the name of the type. Thus, to explicitly convert `p` to its `String` equivalent, we could call `Integer.toString(p)`. We have seen this with `Double.toString()`, too. Remember that `String` is an object type; that means we can use the dot notation to call methods, too.

The Other Literal Value: String

In most computer languages there is a type that is used to hold various sequences of characters. In Java that built-in type is `String` and a `String` literal is a sequence of characters enclosed in double quotes (`"`). Examples include `"simplecomputer"`, `"*.*`", and `"101"`. Escaped characters, characters set off with an initial slash, `\`, permit characters that are difficult to type inside of strings. `"` is a double quote character, not the end of the string literal. `n` and `t` are the *newline* and *tab* characters respectively. A string literal can contain spaces but it cannot span multiple lines in the source code.

The `String` class is special in that it is the *only* class with *overloaded operators*. That is, where `+` normally only works with primitive types, it works for all `Strings`, literal or variables; Java will automatically convert

⁶`String` is a Java defined object type; notice the initial capital letter.

primitive *and* object types to Strings to concatenate them. This is what happens with the values of *p* above. String is a class and instances of it are objects, it is possible to call methods on those instances. String values are also special because they are *immutable* or unchangeable.

```
1 import fang.core.Game;
2 import fang.sprites.StringSprite;
3
4 /**
5  * Demonstrates String functions and String literals.
6  */
7 public class StringValueDemonstration
8     extends Game {
9     /**
10      * Creates local Strings and then demonstrates some String methods in
11      * various labels created on the screen.
12      */
13     @Override
14     public void setup() {
15         final double STRING_HEIGHT = 0.04; // a named constant. Used to step
16                                             // strings down screen
17         double yPosition = STRING_HEIGHT; // represents the y position of
18                                             // the next string sprite
19         String userFirstName = "Claes";
20         String userLastName = "Bos-Ladd";
21         // concatenation
22         String userFullName = userFirstName + " " + userLastName;
23
24         // Constructor takes a string and then the scale.
25         StringSprite msg1 = new StringSprite("userFirstName = " +
26             userFirstName, STRING_HEIGHT);
27         msg1.setLocation(0.5, yPosition);
28         yPosition = yPosition + STRING_HEIGHT;
29         addSprite(msg1);
30
31         StringSprite msg2 = new StringSprite("userLastName = " +
32             userLastName, STRING_HEIGHT);
33         msg2.setLocation(0.5, yPosition);
34         yPosition = yPosition + STRING_HEIGHT;
35         addSprite(msg2);
36
37         StringSprite msg3 = new StringSprite("userFullName = " +
38             userFullName, STRING_HEIGHT);
39         msg3.setLocation(0.5, yPosition);
40         yPosition = yPosition + STRING_HEIGHT;
41         addSprite(msg3);
42
43         StringSprite msg4 = new StringSprite(
44             "userFullName.toLowerCase() = " + userFullName.toLowerCase(),
45             STRING_HEIGHT);
46         msg4.setLocation(0.5, yPosition);
47         yPosition = yPosition + STRING_HEIGHT;
48         addSprite(msg4);
49
50         StringSprite msg5 = new StringSprite(
```

```

51     "userFullName.toUpperCase() = " + userFullName.toUpperCase(),
52     STRING_HEIGHT);
53 msg5.setLocation(0.5, yPosition);
54 yPosition = yPosition + STRING_HEIGHT;
55 addSprite(msg5);
56
57 StringSprite msg6 = new StringSprite(
58     "userFullName.substring(6, 9) = " +
59     userFullName.substring(6, 9), STRING_HEIGHT);
60 msg6.setLocation(0.5, yPosition);
61 yPosition = yPosition + STRING_HEIGHT;
62 addSprite(msg6);
63 }
64 }

```

Listing 5.3: StringValueDemonstration.java

This example demonstrates how to declare `String` variables and assign literal string values to them. It also demonstrates how to build up a `String` using the overloaded `+` operator and assign that result to a variable.

Then some methods of the `String` class are called. `toLowerCase` makes a copy of the `String`, converting all uppercase letters to lowercase (non-letter characters are unchanged); `toUpperCase` is similar but converts all letters to uppercase. `substring` takes a beginning position and one past the last position; note that the character indices in a `String` begin with 0. Thus in the `String` "Claes Bos-Ladd", `substring(6, 9)` returns a `String` containing "Bos" (characters 6, 7, and 8 in the original sequence). The complete results of running this program are shown in the figure below.

Every time you type a literal character string in your Java program (such as "p = " or "Press Any Key to Continue"), Java implicitly creates a new `String` object. In addition to the special handling of literal strings, Java defines the `+` operator as concatenation between `String` values. The last bit of specialness permits `String` to behave like a primitive type because even if two labels refer to the same underlying object, it is not possible to change the characters in the string through *any* reference to it. Thus `String` objects *behave* almost like primitive data types.

Remember, though, that `String` is an object type. That means that `String` objects can be used with the dot syntax to invoke methods on the string. What methods?

Object Expressions

Object expressions are expressions with an object type. Thus `String` expressions qualify. The simplest object expression is an object variable or a call to `new`.

```
new RectangleSprite(1.0, 0.5)
```

This call to `new` calls the constructor for the type `RectangleSprite`. It returns a reference to the newly constructed object. To date we have always assigned the result of this call to a variable. This is only necessary if we want to be able to call the result by name, that is use a label for it. It is legal to use the above expression anywhere a `RectangleSprite` is permitted:

```
addSprite(new RectangleSprite(1.0, 0.5));
```

This would add the new sprite to the game (or `CompositeSprite`). Without a label for the sprite we cannot change its color or scale or rotate or, well, anything.

Using Object Methods

A more complex object expression is the result of calling a method which returns an object type. Consider lines 50-51 in `StringValueDemonstration`, duplicated here:

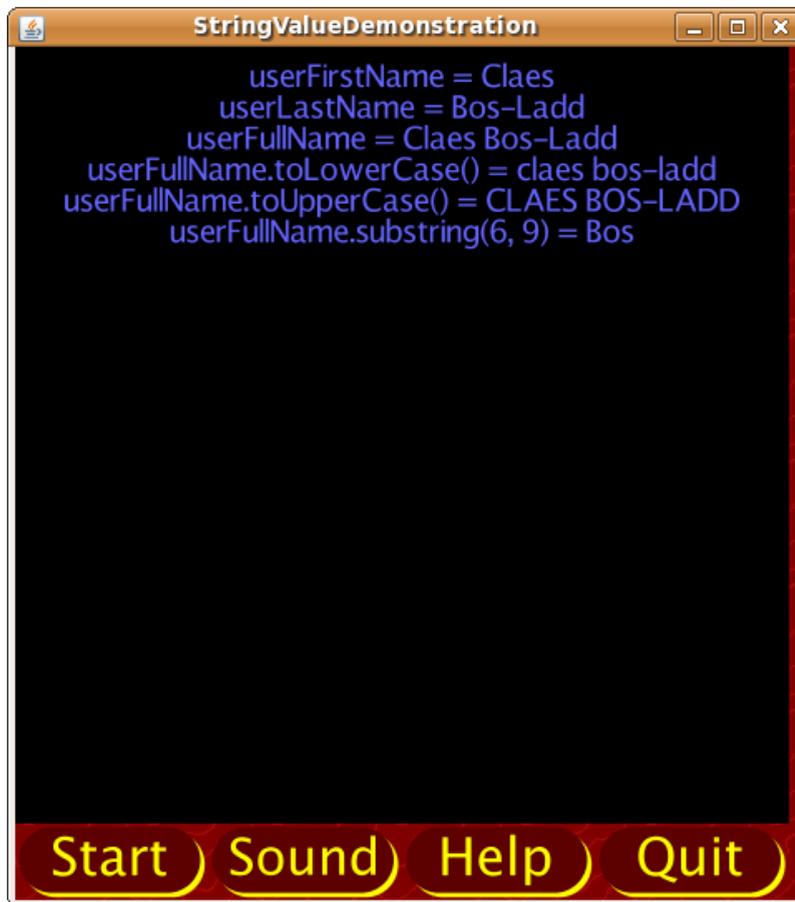


Figure 5.5: Running the compiled StringDemonstration program.

```

45     STRING_HEIGHT);
46     msg4.setLocation(0.5, yPosition);

```

Listing 5.4: StringValueDemonstration: 45-46

The subexpression `userFullName.toLowerCase()` calls the `toLowerCase` method (defined in the `String` class) on the `String` referenced by `userFullName`. A duplicate string is returned with all upper-case letters converted to their lower-case equivalents.

Chaining Method Calls

Because `new` can return an object *and* certain method calls can return objects, it is possible to call methods directly on the results of such expressions. If, in lines 59-61 of `StringValueDemonstration`:

```

59     userFullName.substring(6, 9), STRING_HEIGHT);
60     msg6.setLocation(0.5, yPosition);
61     yPosition = yPosition + STRING_HEIGHT;

```

Listing 5.5: StringValueDemonstration: 59-61

we wanted the substring to be in capital letters we could change the lines to read

```

59     StringSprite msg6 = new StringSprite(
60         "userFullName.substring(6, 9) = " +
61         userFullName.substring(6, 9).toUpperCase(),
62         STRING_HEIGHT);

```

Listing 5.6: StringValueDemonstration: 59-62 (modified)

The middle line *chains* the call to `toUpperCase` onto the results of the call to `substring`.

5.5 Finishing Up SoloPong

Let's consider the public interface for a pong ball. The name `PongBall` makes sense. The public interface has to handle construction, moving the ball (advance) and bouncing the ball off of things. With the bouncing off of things encapsulated in the ball, it is necessary for the ball to count the number of times it has bounced off of the paddle. Thus the ball must also keep track of the score so we have to be able to get and set the volleys count.

The public interface should look something like this:

```

class PongBall
    extends OvalSprite {
    public PongBall(double width, double height,
        double deltaX, double deltaY)...
    public void setVelocity(double deltaX, double deltaY)...
    public void advance(double deltaT, Sprite paddle)...
    public int getVolleys()...
    public void setVolleys()...
    public void incrementVolleys()...
    public boolean isOutOfBounds()...
    public void bounceOffEdges()...
    public void bounceOffSprite(Sprite toBounceOffOf)...
}

```

The constructor and `setVelocity` are straight forward: we need to be able to set the state of the ball, the velocity it maintains. The `advance` method is where that velocity is used to move the ball. The bounce methods are used to keep the ball bouncing off of the edges of the screen, off of the paddle (and any other sprites in the game; see the exercises for suggestions on game variations with multiple sprites on the screen). The final method encodes the rule of the game that the ball goes out of play when it hits the right edge of the screen.

We begin a detailed examination of the code by reading the fields (the velocity) and the constructor.

```

15     * and y-coordinate velocities
16     */
17     private double deltaX;
18     private double deltaY;
19
20     /**
21     * Create a new PongBall. Specify the size and the initial velocity of
22     * the ball.
23     *
24     * @param width width of the ball in screens
25     * @param height height of the ball in screens
26     * @param deltaX initial horizontal velocity in screens per second
27     * @param deltaY initial vertical velocity in screens per second
28     */
29     public PongBall(double width, double height, double deltaX,

```

```

30     double deltaY) {
31     super(width, height);
32     /*
33      * A scope hole: to which deltaX (line 9 or line 25) does the
34      * variable deltaX refer? The closest (furthest in curly braces),
35      * so line 25 How to refer to the field? Use this and a dot.
36      */
37     this.deltaX = deltaX; // set FIELD to the parameter value
38     this.deltaY = deltaY; // set FIELD to the parameter value
39 }
40
41 /**

```

Listing 5.7: PongBall: Constructor

What is line 33 doing? It looks like a call to a method named **super** with two **double** parameters and that is almost exactly what it is. When the keyword **super** appears as the first line in a constructor, it tells Java to call the constructor of the *superclass* or the current class. The superclass is the class which this class extends. Thus line 33 is a call to the `OvalSprite` constructor with two **double** parameters. The two parameters (as we have seen in other uses of `OvalSprite`) are for the width and height of the sprite; we simply pass along the width and height parameters our constructor takes as the parameters for the **super** constructor.

We have never had to do this before because Java defines the concept of a *default constructor*. The default constructor is the constructor with an empty parameter list. If the superclass of one of our classes has a default constructor *and* we do not include an explicit call to **super** on the first line of our constructor, then Java will implicitly call the default constructor *as if* it were the first line of our constructor.

Thus `EasyButton` which extended `CompositeSprite` did not need an explicit call to **super** since an implicit **super**() called the default constructor for `CompositeSprite` which was exactly what we wanted.

Scope Holes

Lines 39 and 40 look strange: the same variable name appears in the line twice, once with **this.** in front of it. What does the comment, mentioning a *scope hole*, mean?

What is scope? Scope is where a variable or method name is usable. A field, like those declared in this listing, is in scope inside the entire class definition (both before and after the declarations). This means, in particular, that `deltaX`, the field, is in scope within the constructor beginning on line 34.

There is a problem, though. The constructor has a parameter also called `deltaX`. A parameter is, effectively, a local variable which is declared just inside the body block of the method on which it is defined. Thus between the curly brace at the end of line 32 and the matching close on line 41, the parameter named `deltaX` is in scope.

This means that between those curly braces there are two *different* meanings of the name `deltaX`. Java has a simple rule for resolving this: whatever definition is *closer* or, deepest in when counting levels of curly braces, wins. Thus the unqualified name `deltaX`, within the constructor, refers to the parameter. This creates a scope hole, a place where the field `deltaX` “should” be in scope but where it is masked by another use of the same name.

But what if we wanted to talk about the field? Remember **this**? The special reference to the current object? Using the dot notation, it is possible to refer to the fields of the current object using **this**. That is exactly what lines 39 and 40 do: the first reference is to the field, the second to the parameter, and the parameter values are used to initialize the fields.

An entire subsection on what a scope hole is; why not just use different names for the parameters and the fields? Up until now we have and it would be perfectly legitimate to do so here.

We create the scope hole to illustrate Java’s rule for handling multiple declarations of the same name at different levels. We also note that many Java programmer’s editors use this construct, naming parameters to the constructor for the fields they will initialize, as a standard; it is important for a Java programmer to be comfortable with the construct and using **this.** to refer to the non-local declaration. Remember that programs

are written for the compiler *and* other programmers; being able to use standard idioms of the language will make your code easier for others to read.

Another use of `this`

```

50     * The first line of a constructor can pass the job to a different
51     * constructor of the current object by calling a method "this".
52     */
53     this(width, height, 0.0, 0.0);
54 }
55
56 /**

```

Listing 5.8: PongBall: The Other Constructor

This listing shows a *second* constructor for PongBall. This raises a lot of questions: Why have two constructors? How does the compiler decide which one to call? What is `this` doing on line 55; it looks like it should be `super`.

This constructor is designed to be like the constructor we use for making OvalSprite, just requiring the user to specify the size of the sprite in screens. This seems convenient. The compiler can decide which one to call by looking at the parameters. If there are four `double` parameters then it calls the first one and if there are only two, it calls the second one. This explains how different constructors for StringSprite were used in StringValueDemonstration above (different number of parameters) and why there are sometimes multiple constructors in the documentation for different classes.

There are two problems for us in writing our second constructor: what value should the velocity get if the user fails to specify one and how do we avoid repeating our selves. The first question has an arbitrary answer; we picked 0.0 along either axis but could have picked any value we liked. The second question is harder.

Constructors exist to make sure that objects are properly initialized before they are used. Among other things, constructors set initial values for the various fields in the class. How can we avoid copying the code that initializes the fields, typing it into two different constructors. This is worth thinking about because if you add another field, you only want to have to remember to initialize it in one place, the place where all the other fields are initialized. And you want to be sure that there is no way to construct class instances without executing that initialization.

Line 54 shows how to avoid repeating yourself in a constructor: if `this` appears as the first line of a constructor, it is a call to another constructor of the same class with the changed signature. This means that `this` must have a different signature than the constructor in which it appears.

Here we pass all the work to the constructor which has the most parameters, specifying the default values in line 55 when we call the constructor.

Refining the Design

`setVelocity` and the `volleys` routines are straightforward; the only tricky thing at all is that both `set` methods use the field name as their parameter name as well. That creates another pair of scope holes where we use `this`. to refer to the field when it is masked by the parameter name.

What does `advance` do?

```

79     bounceOffEdges();
80     if (intersects(paddle)) {
81         bounceOffSprite(paddle);
82         Game.getCurrentGame().setScore(Game.getCurrentGame().getScore() +
83         1);
84     }
85 }
86

```

87 /**

Listing 5.9: PongBall: advance

Using stepwise refinement we can see what happens each frame: we move the ball according to its current velocity. Then we bounce it off of the walls. Then we bounce it off of the paddle. The paddle is a parameter passed into `advance`. Isn't that wrong? According to the documentation, `advance` only takes a **double**. Wait, that is the version of `advance` in `Game`. Since we will call `advance` on our `PongBall`, we can define any signature we like.

By passing in the paddle, the ball can handle all its own bouncing *and* keep track of the volley count. But wait, why is the type of the parameter `Sprite`? Aren't we defining a class, something like `PongPaddle`?

Yes, we are defining such a class but we are using `Sprite` here to make a point: If a method expects some particular class, *any* class extending the expected class or any class extending the expected class and so on, is acceptable. Thus if we expect a `Sprite`, then we can take an `OvalSprite`, a `CompositeSprite`, or even a `EasyButton`. The only limitation when using the super class to pass in parameters is that you're limited to the methods defined in the super class. That is, even if we passed in a `StringSprite`, the formal parameter is only a `Sprite` so you could not use `paddle.setText` because `Sprite` does not have such a method.

Reading the Keyboard

The `PongPaddle` class looks a lot like the `PongBall` class: it extends a `Sprite`, it has a velocity, it has its own `advance` method, and it has a `bounceOffEdges` method. The thing that makes it different is that in the `advance` method, the velocity is only used if an arrow key has been pressed.

Think about that for a moment: for the ball we want to simulate continuous motion so each time `advance` is called, the ball is moved in both the x and y directions:

```
translateX(deltaX * deltaT);
translateY(deltaY * deltaT);
```

We can use the same basic formula to move the paddle except we don't want to move the paddle on *every* call to `advance`; instead we want to check what keys the user has pressed this frame. If they have pressed (and not released) the left arrow key, we want to translate in the x direction by the horizontal velocity of the paddle (`deltaX` because the name makes clear the similarity) times delta time times -1. The -1 is because the x-coordinates get larger as they move from left to right so to move left we want a negative translation. Leaving out the -1 would move the paddle to the right (so we use that with the right arrow key).

As we have seen, reading the keyboard uses the `getKeyPressed()` and `[up, down, left, right]Pressed()` methods of the `Game` class. We now need that information in a different class, a non-`Game` derived class. That means we need to get access to the current `Game` inside `PongPaddle`'s `advance`. The `Game` class has a special method, a **static** method, called `getCurrentGame` which returns a reference to the current `Game`.

static Fields and Methods

What is a **static** method? The short answer is a method which has no **this**. The longer answer is that a **static** method is one which belongs not to an *instance* of a class but rather to the *class* itself.

The difference has two immediate consequences: we can call a **static** method without having called **new** to create an object or instance and inside the method we cannot access *any* fields or methods which are not **static**. The first consequence means that we call **static** methods by prefixing them with the name of the *class* rather than a reference to an instance:

```
Game.getCurrentGame()
```

The second consequence means that **static** is contagious: any method or field you want to use from a **static** method is infected and becomes **static**. Fields, too, can be **static** (we saw **static final** fields in Section 4.3). More discussion of **static** methods, including a modified Java method template, will come later when we write our own **static** methods.

For right now, we will just use the `getCurrentGame` method to get a reference to the current game whenever we need it. So, we can now write the `advance` method for the `PongPaddle` class:

```

47     translateY(-deltaY * deltaT);
48     }
49     if (Game.getCurrentGame().downPressed()) {
50         translateY(deltaY * deltaT);
51     }
52     if (Game.getCurrentGame().leftPressed()) {
53         translateX(-deltaX * deltaT);
54     }
55     if (Game.getCurrentGame().rightPressed()) {
56         translateX(deltaX * deltaT);
57     }
58     bounceOffEdges();
59 }
60
61 /**

```

Listing 5.10: PongPaddle: advance method

Notice that the value returned by the expression `Game.getCurrentGame()` is a reference to a `Game`. It is valid to apply a dot to a reference so `Game.getCurrentGame().upPressed()` is true when the user has pressed the up arrow key (look at the `Game` documentation). Thus these four `if` statements work to move the paddle according to its velocity.

What velocity components should we set to limit the paddle to vertical movement? Vertical movement has a non-zero y-component and a 0.0 x-component. Thus if we set the x-velocity to 0.0, the paddle will only respond to the up and down keys.

Why does the `PongPaddle` even have an x-component to its velocity? For two reasons: sometimes it is easier to solve a more general problem than it is to solve a very specific problem and it almost always a good idea to reuse code you already understand. The ball code is code we have examined and it moves the ball in two dimensions. That code, modified to check on what keys were pressed, works to move the paddle.

The only thing that we need to do differently for the paddle is “bouncing” it off the edge. When the paddle moves a little bit off the edge, we want to move it back immediately. The paddle never moves past the edge so, when we detect that it has, we back it up:

```

71     translateY(0.0 - getMinY());
72     }
73     if (getMaxY() > 1.0) {
74         translateY(1.0 - getMaxY());
75     }
76     if (getMinX() < 0.0) {
77         translateX(0.0 - getMinX());
78     }
79     if (getMaxX() > 1.0) {
80         translateX(1.0 - getMaxX());
81     }
82 }
83
84 /**

```

Listing 5.11: PongPaddle: bounceOffEdges method

The distance we translate is just the amount that the edge of the paddle is past the edge of the screen. We only do the translation if the paddle is past the edge.

Laziness as a Virtue

There is a successful programming language, Perl, which is known for three things among computer programmers: some of the craziest combinations of punctuation as valid variable names (for example, `$_` and `$@` are both regularly used system variables), having more than one way to do *anything*, and for having been developed out of laziness. The inventor of Perl, Larry Wall, praises laziness as a virtue because if you're too lazy to get right to work, you'll think about what you need to do. Perl grew out of a report that Wall had to type up every week with data he had to go across campus to read off of another computer. He automated the process and added parameters so he could change the report format.

Laziness also means that you will look for general solutions that use parameters to specify them. That is, look at the `PongPaddle` advance method. How hard would it be to modify the game to bounce a ball up in the sky while the paddle moved across the bottom of the screen? The paddle is already ready (just set the shape and the velocities). The ball would need a little work.

Of course, some part of the code is specific to the problem at hand. We will end this chapter with a look at the six state handling methods we pretended into existence at the beginning of the chapter.

```

86     *
87     * @return true if we are, false otherwise
88     */
89     private boolean isPlaying() {
90         return ball.isVisible();
91     }
92
93     /**
94     * Whatever state we were in, move us to the moving ball state.
95     */
96     private void beginPlaying() {
97         ball.show();
98         countdown.hide();
99         pressSpace.hide();
100        startBall(ball);
101        setScore(0); // no volleys yet
102    }
103
104    /**
105    * Are we in the waiting to countdown state?
106    *
107    * @return true if we are, false otherwise
108    */
109    private boolean isWaitingToCountdown() {
110        return pressSpace.isVisible();
111    }
112
113    /**
114    * Whatever state we were in, move us to the waiting to countdown
115    * state.
116    */
117    private void beginWaitingToCountdown() {
118        ball.hide();
119        countdown.hide();
120        pressSpace.show();
121    }
122
123    /**

```

```

124  * Are we in the counting down state?
125  *
126  * @return true if we are, false otherwise
127  */
128  private boolean isCountingDown() {
129      return countdown.isVisible();
130  }
131
132  /**
133   * Whatever state we were in, move us to the counting down state
134   */
135  private void beginCountingDown() {
136      ball.hide();
137      countdown.show();
138      pressSpace.hide();
139      countdown.setTimer(COUNTDOWN_SECONDS);
140      countdown.startTimer();
141  }
142
143  /**

```

Listing 5.12: SoloPong: State Handling

Only one of the three sprites, `ball`, `pressSpace`, and `countdownTimer` is visible at any time in our game. `ball` is visible when the player is playing. `pressSpace` is a `StringSprite` telling the player to press space to start the game; when it is visible, the program is waiting to countdown. `countdownTimer` is only visible when it is counting down. Note that invisible sprites will still report intersection, one with another or with visible sprites. This can be a problem in some games but since two of the three sprites are never tested for intersection, we are fine.

The three `is<StateName>` methods just return the visibility of the patron sprite of the state they name. The first three lines of the `begin<StateName>` methods are the same: they set the visibility of the three patron sprites to match the state they are beginning. After the first three lines, `beginCountingDown` sets the countdown time and starts the timer. `beginPlaying` sets the score (volley count) to 0 and uses `startBall` to start ball.

The use of a method, `startBall` and a parameter, the `ball`, is done with an eye toward having more than one ball in play. It also takes advantage of the fact that `ball` is a reference to a `PongBall` object so that the location of that object can be set in the method.

```

75      theBall.setLocation(randomDouble(0.2, 0.4), randomDouble(0.3, 0.7));
76  }
77
78  /**

```

Listing 5.13: SoloPong: initializeBallLocation

The actual `startBall` method, shown here, starts the ball moving down and to the right at a 45 degree angle. It then randomly places the ball in a rectangle near the middle of the left side of the screen; this makes it unlikely that leaving the paddle in one place will keep the game going across multiple restarts of the ball.

5.6 Summary

Delegation

Delegation is the creation of new named rules. In Java this is done by defining methods. The four parts of the signature remain the access level, the return type, the name of the method, and the parameter list.

The *access level* can be **private**, **protected**, **public**, or nothing. **private** means that access to the method (or field) is limited to just the class in which it appears. **protected** means the same as **private** except for classes extending the current class; child classes of the current class can access **protected** fields and methods. **public** means that any class which has a reference to the current class can call the method (or access the field). Having no access level means that the method or field is accessible to the *package*; for the moment, package can be considered the folder containing the source code file.

There are two different parameter lists: the *formal parameter list* is the one in the signature, a list of types and names; the *actual parameter list* appears when the method is called and is a list of expressions. The formal parameters behave like local variables which are initialized with the values of the actual parameter list.

In Java parameters are passed by *value*. This means the expression's result is copied into the formal parameter; changes made to the formal parameter inside the method do not modify the actual parameter.

An apparent contradiction is references to objects; the reference is copied into the formal parameter but both the formal and actual parameter refer to the same value.

Delegation is how top-down design can be applied. By assuming that the next lower level of solutions exist, a method can be written. Then, the simpler problems represented by the design of the sub-methods are approached by defining the methods.

Scope

The *scope* of a variable is where the name is visible. Fields are in scope from the opening curly brace of the **class** to the closing curly brace. Formal parameters are in scope throughout the block defining the body of the method. All other variable declarations are in scope from the line where they are declared until the end of the block in which they are declared.

Chapter Review Exercises

Review Exercise 5.1 Would it make sense to have a constructor for `PongBall` which only takes an initial velocity? Why or why not? If it makes sense, can you write it?

Review Exercise 5.2 Assume you write a `makeAllTheSprites` method which takes no parameters.

- How would you call `makeAllTheSprites` from `setup`?
- Describe what happens to the running version of `setup` when `makeAllTheSprites` is called.

Review Exercise 5.3 Describe the *public interface* of a class.

Review Exercise 5.4 `cr:Rules:ScopeHole` What is the type of `x` at each of the lettered lines?

```
public void someMethod(double x) {
    // (a)
    if (x > 100.0) {
        String x = "Hello, World!";
        {
            int x = 10;
            // (b)
        }
        // (c)
    } else {
        // (d)
    }
    // (e)
}
```

Review Exercise 5.5 Using the code in the previous exercise, describe where there is a *scope hole* for *x*.

Review Exercise 5.6 Consider `Game.getCurrentGame()`.

- What is the *type* returned by this method?
- What does it mean that `Game` is the name of a *class* rather than a reference to an instance?
- Can you write the signature for `getCurrentGame`?

Review Exercise 5.7 What is the value of *k* at the lettered lines?

```
public void alpha(int k) {
    // (a)
    k = k + 100;
    // (b)
}

public int beta(int k) {
    // (c)
    return k * 2;
}

public void setup() {
    int k = 13;
    // (d)
    alpha(k);
    // (e)
    int j = beta(k);
    // (f)
    k = j;
    // (g)
}
```

Review Exercise 5.8 What color is the rectangle on the screen after this code executes?

```
public void aleph(RectangleSprite rect) {
    rect.setColor(getColor("blue"));
}

public RectangleSprite beth(RectangleSprite rect) {
    rect.setColor(getColor("black"));
    return rect;
}

public void setup() {
    RectangleSprite rs = new RectangleSprite(0.5, 0.5);
    rs.setLocation(0.5, 0.5);
    addSprite(rs);
    // what color if we stop here?

    aleph(rs);
    // and if we stop here?
    rs = beth(rs);
}
```

Review Exercise 5.9 What method does `Game` provide to examine the player’s mouse position? What method does `Game` provide to examine the player’s keyboard?

Programming Problems

Programming Problem 5.1 Write a computer program that creates a tenth of a screen red square that bounces around the screen. When the player clicks the mouse on the moving square, the color should change to cyan and then, when it is clicked again, it turns back to red. Also keep track of the number of times the user successfully clicks on the square (don’t bother keeping track of misses).

This problem should be implemented *incrementally*. Three levels of refinement might be:

- (a) Make a program with a red bouncing square. You will need a reference to the bouncing square that is visible to multiple routines; what should you be thinking about *where* to declare the label for the bouncing ball. (*Hint: Look at `PongGame` for some guidance on creating a bouncing object.*)

Why? Animating a bouncing `RectangleSprite` seems to have the simplest subset of the functionality required by this program. So this seems to be the smallest program that does *something* along the path to a program that fulfills the requirements described above.

- (b) Add a score. This would require a numeric component that is visible from multiple methods (`preCodeHook`, some display building method, and the mouse listener (see below)). Also, add label on the screen so the player can see their square
- (c) Modify your bouncing program so that the bouncing square has a `MouseListener` listener for pressing/releasing the mouse. add a

Playtesting. Though this game is very, very simple, it offers an interesting chance to do some playtesting to balance difficulty and enjoyment of the game. First, think about what you could do to make the game easier.

The two ideas that come to mind are to make the target bigger or to make it move more slowly (or some combination of both). For the purposes of this exercise we will leave the size fixed and manipulate the speed.

Where is the *velocity* of the moving square set? Change the speed to one half of its current value and play the game. It will be easier; is it more fun? It probably will be if you found the previous speed too difficult. Try doubling the velocity from the original value. Is the resulting game more fun? Or is it frustratingly difficult?

Now that you know the fun/challenge ratio for three different settings, you can work out the setting that makes the game most enjoyable for you. Note that in the *real-world* playtesting involves a lot more testers and an average of the results typically sets the difficulty level of the final game. Think a little bit about how you could accommodate gamers of different abilities and keep an eye on the **Playtesting** sections in other Programming Problems.

Programming Problem 5.2 Going back to `OneDie.java`, can you add an animation to the dice? When the dice are rolled, they should start randomly changing the visible face once every quarter of a second and should “roll” for two seconds (both values should be fields and should have `get/set` methods).

`EasyDice` will have to be modified so that it asks the dice if they are still rolling. If they are, ignore user input. This implies the following changes to the public interface for `OneDie`:

```
public void setFaceTime(double secondsPerFace) ...
public double getFaceTime() ...
public void setRollTime(double secondsToRoll) ...
public double getRollTime() ...
public boolean isRolling() ...
```

Make sure you provide good comments for the methods (and the fields that you add).

Programming Problem 5.3 Start with `SoloPong.java`.

- (a) What changes would be necessary to handle two balls at the same time? How would you be able to start the balls so that their arrival at the paddle is somewhat staggered?
- (b) Modify the game so that two balls are started whenever one ball is started. Also, only change the state of the game to waiting to start play when *both* balls go out of play.
- (c) Modify the game so that the second ball comes into play whenever the volley count (with one ball) goes to 10. That is, after I get 10 volleys, a new ball appears so I can get volleys twice as fast.

Playtesting. Look at the two two ball variations spelled out here. Try each a few times. Which seems to make for more interesting gameplay? Why do you think that is? Would it make the game more interesting to have the two balls move at different velocities?

Programming Problem 5.4 Start with `SoloPong.java`. Modify `PongBall` so that after each volley it speeds up by 5%. Does this variation make the game more fun?

Programming Problem 5.5 Add a target object to the game. Create a class which when struck by the `PongBall` disappears. Disappearance is handled either by moving the object off of the screen or calling `removeSprite(<spriteToRemove>)`.

Notice that if you have one object that can disappear when struck, you can have an arbitrary number of them and you could build a *Breakout* clone.

Components Meet Rules: Classes

Abstraction, as a problem solving technique, means the creation of a new type. We have created our own classes, by extending FANG classes such as `Game` and `CompositeSprite`. This chapter will explain, in much detail, much of the nuance of designing your own type. It also presents the first *networked* game, finally exploiting the “N” in FANG. Finally, it also introduces Java’s standard output stream, a way to print messages in the window where Java is running. The output is more primitive than that provided by `StringSprite` but it is easier to set up and can be used for simple debugging.

6.1 Playing Together

From the start FANG was designed with an emphasis on networked games, games with multiple players playing at different computers, perhaps in the same room and perhaps half way across the globe. Getting input from multiple players is almost identical to getting input from a single player. FANG games can run inside of Java-enabled Web browsers *and* they can run on the desktop; you can easily distribute the game to multiple players on computers of different capabilities.

6.2 Network Programming with FANG

To focus on the networking aspects of our game, we will implement one of the fundamental games most people remember from their childhood: Tic-Tac-Toe.

TicTacToe is a game of position. The board is a 3 × 3 grid of squares. Two players alternate turns, each placing their symbol, an X or an O, in one of the squares. Play continues until one player has created a line of three of their symbol horizontally, vertically, or diagonally on the board or there are no more spaces on which to move.

If a player succeeds in getting three in the row, they win. If the game ends without three in a row, the game is a tie or, in Tic-Tac-Toe terms, a “Cat’s Game”.

We could create a solitaire version of *TicTacToe* where a single player places one of the symbols and an artificial intelligence of our devising places the other symbol. While *TicTacToe* is simple, if we recall there are 3⁹ different board positions. Determining a strategy (rules for actually playing or trying to win at a game; see Chapter 1) for such a game would not be easy.

Easier, especially given FANG’s networking capability, will be building a multi-player version of the game where each player runs a separate copy of our *TicTacToe* program. Then, the two copies of the program will communicate so that play alternates between the two players and the game will detect when one of the two wins.

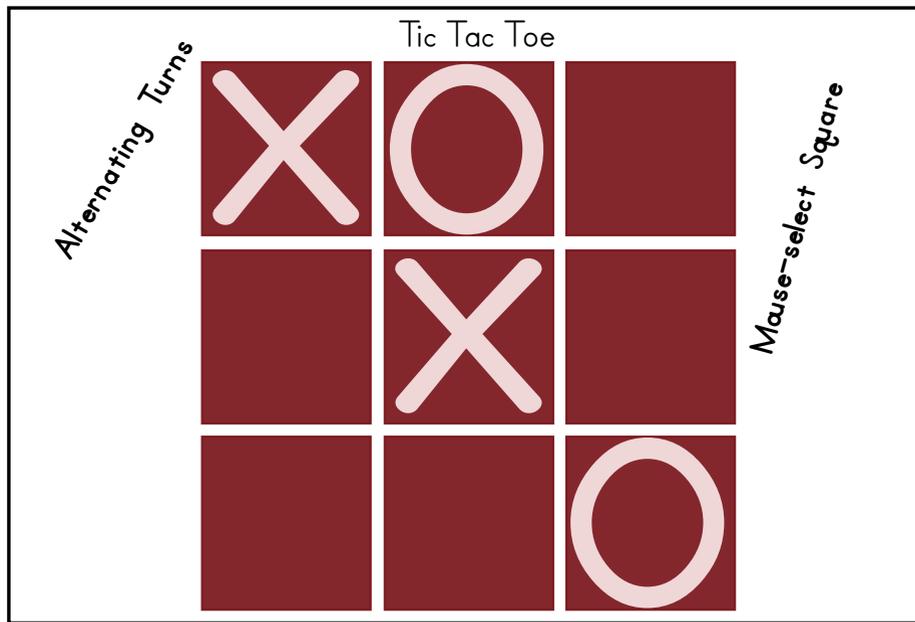


Figure 6.1: TicTacToe game design diagram

Sharing State: Where's the Game?

When two (or more) players share a computer game there are several interesting questions. One is how do the programs communicate at all, another is given that they can pass messages, what messages do they pass, and finally, where is the game?

A *process* is a running program¹. Communication between processes is known as *interprocess communication* (IPC). IPC is handled by the *operating system* with different systems supporting different kinds of IPC.

One particular standard for communication, *TCP/IP* or the *Internet protocols*, has emerged as a very widely supported collection of protocols which work on many different hardware platforms and operating systems. One of the reasons they work on multiple platforms is that they are built using *layers of abstraction* where each lower layer advertises a public interface and each higher layer implementing its portion of the communications process using only the public interface.

The Internet protocols provide a reliable mechanism for transferring messages from one process to another on the same or different computers, assuming the computers are attached to the Internet. The Internet is a network of networks implementing machine addressing, message routing, and the actual delivery of messages from machine to machine. We will talk about the details of the Internet protocol when we talk about streams in Chapter 11.

If we can pass messages using the Internet protocols, what should our processes say to one another to run our game? The answer to that depends a lot on the answer to the third question: where does the game live?

If multiple processes are cooperating to play a game, the game state could live one of three places:

1. In one process
2. Part of the state in each process
3. All of the state in each process

¹As opposed to the bytecode or machine code sitting in a file which is called a program. We are only this precise when it is absolutely necessary.

If all of the state lived in one process that process would be the *server* process. All other processes would have to provide requests to that process for doing what their player has tried to do with the input and the server would have to decide what happened and communicate the results back to all the other processes. The other processes in this model are called *clients*. If the terms client/server sound familiar, it might well be because a Web server keeps a collection of Web pages and a Web browser (client) requests pages from the server and the server decides what pages the client gets.

If part of the state resides in each process without a server process, then each process must communicate its portion of the state to all other processes playing the game. Since every process is symmetric, that is every process has the same communications needs and abilities, they are often referred to as *peers* and a configuration like this is a *peer-to-peer* configuration. Peer-to-peer communications patters are seen in instant messaging programs and file sharing systems such as Bittorrent.

The final configuration requires that the input fed to each process be the same. That is, if one player presses “A”, then all of the processes must process that keystroke in the same order relative to all other input. This means the input needs to be shared from process to process and they must all be ordered. This is most easily accomplished using a client/server model where all input goes to the server and the server sends the combined input stream out to all clients.

This last model, where all games simply process the same sequence of input to remain in synch has an obvious vulnerability: if for some reason my game did something different with the input, my view of the game would not match yours. If I rewrote my game to give me hidden knowledge (say seeing your cards in a card game), then I would be cheating.

FANG uses this third method because the input stream to FANG games is typically fairly small so sharing one or more such streams across the network is not too bandwidth hungry. An interesting note is that you have been using this model with every game you have written. In order to handle multi-player games well, even single player games create a server to handle input and then keystrokes and mouse moves are forwarded to that server and then bounced right back to the single-player game.

Distributing the Game Loop

How is distributing the input handled in the video game loop? We will review the game loop and then discuss what happens in each players’ process.

Reviewing: Display; Get Input; Update State

The game loop has three parts:

- Display the current game state
- Get user input
- Update game state according to current state and user input

When we distribute the game across multiple processes, each copy of TicTacToe runs the same game so FANG, in each process, runs the video game loop. The only real difference from our point of view is that the user input can come from the local player or from any other player connected to our game. So long as all copies of the game see the same input from all players in the same order, the update action will be the same at each game.

Sharing Information

We will define `GameTile` a lot like `EasyButton` (review Section 4.1 if you need to). Thus there will be an `isPressed` method which can be called from the game’s `advance` method to determine whether or not the user has clicked the mouse in a given square.

The difference this time is that rather than using the `getClick2D()` method of the current game (no parameters between the parentheses) we will use the `getClick2D(playerID)` where `playerID` is an `int` which tells the game which of the multiple players we are interested in. The first player (the only player in single-player games) is player 0; the other player in a two player game is player 1.

```

124     Location2D mouseClicked = Game.getCurrentGame().getClick2D(playerID);
125
126     if (mouseClick != null) {
127         if (intersects(mouseClick)) {
128             return true;
129         }
130     }
131     return false;
132 }
133 }

```

Listing 6.1: GameTile isPressed

The method, `isPressed` looks almost identical to `isPressed` in `EasyButton`: it gets the current game, fetches the mouse click location, and checks if the mouse location intersects this sprite. The differences are the parameter, `int playerID` on line 124 and then the use of the parameter when calling `getClick2D` in line 126. This means that our game cares about which player clicked the button. In the game we will only call `isPressed` with the player number of the current player. If it is X's turn, only player number 0 will be able to click; alternatively, if it is O's turn, only player number 1 will be able to click.

We will see how the player numbers are assigned and how a multi-player game is started later in this chapter. First we will look at *abstraction* as a problem solving technique in Java.

6.3 Abstraction: Defining New Types

Java classes permit the construction of *abstract data types*. An abstract data type consists of three parts: a public interface, a private implementation, and a strict limitation of interaction with the type to the public interface. The public interface, as discussed throughout the book, is the collection of **public** methods, **public** fields², and any types which are exposed from the inside of the class. The private implementation is everything else, in particular what we have been calling *state* or the *attributes* of objects of the class. The admonition to limit interaction with objects of a given type to the public interface means that the author of the class can define the operations (or rules) applicable to objects of that class; the public interface operations can enforce any constraints the author requires, constraints which could be violated if users of the class directly manipulated the private implementation.

What is a Type?

What is a type? One answer, for beginning Java programmers, is that a type is a **class**. This is a tempting equivalence because **class** is the way users define their own types. Type is a larger concept, however. The plain old data types are, in fact, types while they are not classes.

Abstract Data Type

An *abstract data type* is a type with three parts:

1. A *public interface*. The collection of operators which users of the type may call along with any publicly accessible fields and exported internal types.
2. A *private implementation*. Also known as a “layout in memory”. For classes this means the private fields and methods and internal types. For non-classes the idea of a layout in memory makes more sense; the bits inside the POD type have an interpretation defined by the language designers and that is the implementation.

²Typically considered bad practice for *mutable* classes. A mutable class is one which can be changed after it is constructed.

3. Interaction is limited to the public interface. No code using the type is permitted to access the private implementation directly; all access to the value of the type is through the interface.

Java's `double` type is an abstract type. The public interface includes declaring `double` variables, the standard arithmetic operators (+, -, *, and /) which work with `double`, and the cast operator, `(double)` which forces Java to treat a given expression as having the type `double`.

What is the private implementation. We don't know. In fact, we don't *want* to know. If we learn the internal representation of a floating point number and we decide to manipulate the representation directly (ignoring point 3 above), we must take responsibility for any constraints that the representation has. Otherwise our manipulation might set a `double` in memory to a value which cannot be used to encode a `double`.

Alternatively, consider a `RationalNumber` class. A rational number is a number of the form $\frac{p}{q}$ for integers p and q . If we kept two `int` fields, p , and q , what constraints, if any, would there be on what values of the two fields would represent *legal* rational numbers?

A rational number cannot have a denominator of 0. Thus q can never be 0. Yet, if we made the fields `public` (the equivalent of letting programmers directly manipulate the bits inside a `double`), any idiot could put the line `rational.q = 0;` in their code and change the previous value of `rational` to an illegal `RationalNumber`.

If we "know", by examining the public interface and limiting access to the public interface, that q cannot be 0, then we can freely divide by q whenever we want without causing a divide by zero exception.

Alternatively, we could make sure that every time, just before we ever divide by q , we check whether q is 0. This would work so long as *every single programmer* who worked on the class understood and followed this convention. Missing the check even once will cause our program to harbor a serious bug. An *intermittent* bug, one which does not fire every time the program is run, is always more difficult to find because you must determine when it happens. Then, by repeating the *reproduction steps* you can trigger the bug; only after you have reliable reproduction can you begin to determine the cause of the error.

Our Own Types

We have looked at public interfaces, the way of specifying the top-level interaction of a new type. The public interface determines where we begin step-wise refinement for the class.

The point of *abstraction* is dividing a problem into a group of cooperating types. Each type is responsible for one particular aspect of the problem. The types are then combined to create a working program. For our `TicTacToe` program, the primary class we will use along with the game itself is the `GameTile` class. As mentioned above, we will be able to select which player gets to click on a square at any given point. We will also extend the class from the `EasyButton` class by looking at how to define *named constants*, values which set constant values for use in a class.

Structure of class Files

The following template extends our previous `class` definition template by defining several sections within the code. Note that these sections can be mixed as the programmer sees fit; we provide this ordering to standardize how classes are laid out and make it easier for a programmer to orient themselves in our code.

```
public class <ClassName>
  extends <ParentClassName> {
  <static definitions>
  <field definitions>
  <constructor definitions>
  <method definitions>
}
```

What is a *static* definition? Static fields and methods are fields and methods defined for the *class* and not for individual *objects* of the class. That is, there is a single copy of the field (or method) for all instances of our class no matter how many instances there are.

Looking at `GameTile`, we see two `static, final` fields:

```

22  /** the constant (final), class-wide (static) Color for the text */
23  public static final Color DEFAULT_CONTENT_COLOR = Palette.getColor(
24      "lavender");
25
26
27  /** The background of the tile */

```

Listing 6.2: GameTile named constants

The first *modifier* before the field type, name, and value is **public**. Didn't the entire section on abstract data types spell out that **public** and fields do not mix? What makes these fields special? The *second* modifier.

final is a keyword meaning that the value of the field, variable, or parameter is *constant* after it is set for the first time. Lines 22-23 declare and assign a value to the field `DEFAULT_COLOR`; it is not legal to use `DEFAULT_COLOR` on the left-hand side of an assignment statement anywhere else.

What use is a **final** field? The **final** keyword is a comment that the compiler can enforce. The field is marked as constant (immutable), documenting your intent that it not change in a way that the compiler can read. If a constant value is used a lot of different places, then it is a good idea to name it so that you *do not repeat yourself*.

The DRY principle is only half of the benefit of having a **final** field. If you look through a moderately long class definition and find that it always initializes its location to (0.5, 0.5), how would you change the code to have it start at (0.0, 0.0)? You would have to look at all calls to `setLocation` and, in turn, at all uses of the number 0.5. You would have to figure out whether any of those uses is one that needs to be changed. Further, you would have to make sure you didn't change any of the uses of 0.5 that were not used for initial location setting.

With a named **final, static** field, you would document what the 0.5 is *for* and give someone modifying your class a single point to modify:

```

public final static double INITIAL_X = 0.5;
public final static double INITIAL_Y = 0.5;

```

So, what is **static**? It means there is only one copy of the field for all instances of the class. Consider the field that is set using `setColor` for a sprite. The field is called `color` and each sprite has its own instance of the field. That only makes sense since if you `setColor` on some `StringSprite` you do not expect the color of all other `StringSprites` on the screen (or `PolygonSprites` or `RectangleSprites`, etc.). Thus `color` is *not static*. The default color for the text and background of the `GameTile` does not need to be stored more than once. Changing the default color should change the starting color of all `GameTile` created after the change (hypothetical change here; the field is also **final** so it doesn't change).

This field is safely **public** because it is **final**; no one can change it. It is **public** so the game using `GameTile` can access the default color if it is interested in using the color for something else.

How do you reference a **static** field (or a **static** method)? Regular fields/methods must appear after a reference and a dot (or, in a method *inside* a class, without anything, thereby implying **this**. in front of them). A **static** field uses the name of the class followed by a dot: `GameTile.DEFAULT_COLOR`.

Lines 22 and 26 show examples of using a **static** method: the `Palette` class defines several `getColor` methods as **static** methods. Since the value of the **final, static** is set before the program really starts running, we cannot access `getCurrentGame`. Fortunately, `Game` just passes calls to `getColor` on to the `Palette` class in any case so we can just use `Palette.getColor` directly.

Constructing Instances of Our Type

The *constructor* for our class must initialize all *instance* fields, the “regular” fields we have been using all along. The constructor signature is special in two ways: it has no return type and it has the class name as its name. The return type makes no sense because the constructor is initializing an object of the type it constructs.

When you call **new**, Java looks at the type being constructed. It then sets aside enough memory all together to hold all the fields of an instance of that type. Thus the more fields you have, the more space an object of that type takes in memory.

The constructor for the class is then called with **this** set to refer to the newly allocated memory. The memory is somewhere in the RAM being used by the Java virtual machine. The constructor then initializes all the instance fields (which either explicitly or implicitly are prefixed with **this**. and are thus located inside the memory Java just allocated).

```

29
30  /** the visible content of the tile */
31  private final StringSprite displayContent;
32
33  /** Current content of this tile */
34  private String content;
35
36  /**
37   * Construct a new GameTile: content = ""
38   */
39  public GameTile() {
40      background = new RectangleSprite(1.0, 1.0);
41      background.setColor(DEFAULT_COLOR);
42      addSprite(background);
43      content = "";
44      displayContent = new StringSprite();
45      displayContent.setLocation(0.00, 0.075);
46      displayContent.setColor(DEFAULT_CONTENT_COLOR);
47      addSprite(displayContent);
48  }
49
50  /**

```

Listing 6.3: GameTile constructor

The three fields of `GameTile` are shown in lines 30-37 in the above listing. They are based on those in `EasyButton`: a `RectangleSprite` for the background of the square, a `StringSprite` to show the content of the square and a `String` which holds the content. Note that two of the fields are marked as **final**. That is, after each is initialized in the constructor they may never be assigned a different value. This is a promise to the compiler *and* to programmers reading this code. The value will not change after the first assignment. Thus **final** serves as compiler enforced documentation.

In the interest of simplicity we have not used **final** before this point for our fields. It is considered good design practice to use **final** whenever possible. It simplifies reasoning about when things might change.

Key thing to notice about **final**: while `background` is a **final** field and the *object* it references cannot be changed, the non-**final** fields of the object referred to by `background` can be changed.

```

102  }
103
104  /**

```

Listing 6.4: GameTile setColor

The `setColor` method calls `background.setColor`; thus the location *referenced* by `background` isn't changed but the color field, stored in the contiguous space allocated when **new** was called (see line 42 above), *does* change.

The `content` field is not **final** because we provide a `setContent` method which changes the content.

```

68      content = content.substring(0, 1);
69  }
70  this.content = content;
71  setText(content);

```

```

72 }
73
74 /**

```

Listing 6.5: GameTile setContent

Notice that we use the ability to *validate* the value being set in setContent (the reason we don't want to make regular, mutable fields `public` is we can't validate). We make sure if the value has more than one character we chop it down to its first character. This makes sure the displayed value is large and fills the square.

Documenting our Types

A type is documented by the combination of header comments, typically written in JavaDoc style, and any necessary in-line comments. On large projects, there may well be external documentation as well, design documents, requirements documents, and component tests. At a minimum you should include a header comment for every class, every field, and every method defined in the class.

The JavaDoc program will, by default, extract header comments for `public` and `protected` elements, generating a linked collection of HTML pages which can be viewed with a Web browser. The use of the JavaDoc tool is beyond the scope of this book but you should be aware of its existence.

Document Intent

Over and over you have been admonished to document your intent. This means that your comments, header and otherwise, should talk to someone who already knows Java as well as you do. The comments should explain why the class/method/field exists as well as why someone would want to use it.

Documentation is another level of abstraction which a competent programmer keeps in mind: dividing solutions into multiple cooperating classes and using step-wise refinement to develop methods are language ways of taming complexity with levels of abstraction. The comments on a class, method, or field give you a slightly higher level to explain the function of your class in a natural language. The documented intent serves as a table of contents for the bit of code it comments on, helping the reader find their way around your code.

6.4 Finishing the Game

What does advance for TicTacToe look like? Each frame we need to check if the current player moved. If they haven't moved, then advance is done.

If the current player has moved, then we must check if they won the game. If they have, the game is over. If they haven't won it is possible that the game is now unwinnable; if it is a cat's game, the game is over. If the game is not over, then we switch which player's turn it is.

```

293     if (isPlayerMoved()) {
294         if (isWinner(currentPlayerSymbol)) {
295             handleWin(currentPlayerSymbol);
296         } else if (isCatsGame()) {
297             handleCatsGame();
298         } else {
299             nextTurn();
300         }
301     }
302 }
303 }
304 }

```

Listing 6.6: TicTacToe advance

The first thing to notice is that there are a lot of lines in `TicTacToe.java`; this is the last method in the class but more than 300 lines makes this the longest program we have looked at so far. We will find that there are sections of code that are repetitious (and we are going to wish for some way to apply DRY; that is left as an exercise in Chapter 7).

The second thing to notice is that the names of the methods called by `advance` serve, to an extent, as documentation for how it works. That is, reading the code as written makes much more sense than if the methods were just `methodA` through `methodE` (`isGameOver` is provided by FANG so we don't get to pick that name). While this author claims there is no such thing as completely self-documenting code³, this code is fairly simple to follow because of the names chosen.

One more thing to notice. Because `advance` is an entry point for our top-down design of a Game-based program, this is at a fairly high level of abstraction. This is evident in the complete lack of all but one field name here. The details of how the methods work was put off until they are written. The only requirement this has is that we be able to know what symbol belongs to the current player.

The Tic Tac Toe Board

How should we represent the `TicTacToe` board? It is 9 tiles, three rows of three. We have a `GameTile` class. Thus the board is nine variables of type `GameTile`. The tile know how to determine if they have been clicked. When checking for a player's move, we will ask each square if it has been clicked by the current player. If so, they moved. If not, go on to the next square.

Naming fields becomes annoying when there are 9 variables which are pretty much the same except for where they are in the grid.

```

19  private GameTile t31, t32, t33;
20
21  /** Which character, X or O, goes next? */
66  currentPlayerID = 0;
67  currentPlayerSymbol = "X";
68  turnsTaken = 0;
69  t11 = makeGameTile(0.17, 0.17);
70  t12 = makeGameTile(0.50, 0.17);
71  t13 = makeGameTile(0.83, 0.17);
72  t21 = makeGameTile(0.17, 0.50);
73  t22 = makeGameTile(0.50, 0.50);
74  t23 = makeGameTile(0.83, 0.50);
75  t31 = makeGameTile(0.17, 0.83);
76  t32 = makeGameTile(0.50, 0.83);
77  t33 = makeGameTile(0.83, 0.83);
78  setTitle(currentPlayerSymbol + "'s Turn: TicTacToe");
79  }
80
81  /**

```

Listing 6.7: TicTacToe setup

The nine `GameTile` are named by their location in the board (see lines 19-21). These lines show that when declaring fields (or local variables), the type name can be followed by a comma-separated list of names. Thus the template for field and variable declarations should be:

```
<fieldDeclaration> := private <TypeName> <fieldName1> [, <fieldNamei>]* ;
```

```
<varDeclaration> := <TypeName> <fieldName1> [, <fieldNamei>]* ;
```

³Advocates for self-documenting code claim that with properly chosen field and method names, it is possible to write computer code that needs no comments to be understood. The earlier argument that comments serve as a table of contents or outline to the code to which they apply seems to say that a good level of comments will always make code easier to follow. This does not excuse *poor* naming conventions, it just says that good naming and good commenting compliment one another to make code readable.

This permits the declaration of an arbitrary number of fields (or local variables) of the same type in a single statement. We will use one declaration per line unless there is a reason not to. The game tile are declared on three lines because their layout in the declaration serves to document where each entry is on the board.

We need to initialize nine tiles; to avoid repeating ourselves, we define the `makeGameTile` method. It takes the center location for the tile as parameters; the center location is the only attribute which changes from tile to tile. The constructor, size setting, and sending the parameters to set the location, and adding the sprite to the game are the same for every tile. `makeGameTile` returns the reference it gets back from `new` so that the nine variables can be initialized (lines 71-79). Lines 67-70 should be clear: the three other fields of the class are a turn counter, a `String` with the symbol for the current player, and a `currentPlayerID`, the FANG ID number for the player whose turn it currently is.

Starting a Multi-player FANG Game

How does FANG know a game is multi-player? Given that a game is multi-player, how do multiple players connect to the server? And who starts the server anyway?

```

36     setNumberOfPlayers(2);
37     // players is a protected field above us (with no set method)
38     players = 2;
39 }
40
41 /**

```

Listing 6.8: TicTacToe constructor

This is the first `Game`-derived class for which we have defined a *constructor*. Typically we have been able to use the standard settings for everything in a `Game` or we have modified the value inside `setup`. Unfortunately, by the time `setup` is run, the number of players in the game has already been determined. Thus we must hook into the `Game` earlier in its life.

FANG always uses the *default* constructor when building a `Game`-derived class. Thus we must build our own default constructor. Line 36 is included to remind us that any constructor must call some constructor of the parent class. Here we are calling the default constructor of `Game`. This is the same constructor that would have been called if line 36 were omitted or commented out; the line is here to make clear what happens. After the constructor comes back we set two fields of `Game`: `numberOfPlayers` and `players`. Both are set to 2 to make our game require 2 players.

When you run the program, FANG notices that it is a multi-player game before calling `setup`. The game brings you into the multi-player *lobby*.

The lobby has three text fields and a button. From the bottom up, the **Connect & Start Game** button connects to the server described by the next field up, the **Name of game server**. The author's machine is named "BigRedOne" which is why this field contains that name automatically. When running an applet, the game server is assumed to be the Web host that served the applet; when running an application, the server is assumed to be the localhost. **Number of players** indicates the number of players expected to play the game. While it can be changed from the lobby, `TicTacToe` cannot handle a different number of players. Finally, **Name of this game** is a name you can provide for the session you are running. This is so that multiple sessions of a given game can use the same server and still be told apart.

Since our game is `TicTacToe`, the session name will be "t-cubed". If there are already sessions of this game running and waiting for players on the server, the drop-down box will contain the name of the sessions already running. After typing in "t-cubed" and pressing the **Connect & Start Game** button, the first player sees a waiting screen.

When a second player starts a copy of the program, they will see the same lobby. They can direct their program to any machine they wish by typing the IP address or name into the **Name of game server** field. The screenshot shows the "BigRedOne" name for the game server. Having said that, the list of sessions running the current game contains one entry, "t-cubed". If that name is picked and the second player presses the button, two players connect to the same session, a session which requires two players, and the game begins.



Figure 6.2: Lobby for TicTacToe

The initial game shows nine `GameTile`. Either player may press **Start** and then the game begins. The first player who chose the name for the session is player 0; the second player to join the session is player 1 and so on for any other players. This permits the game to know how to allocate score and keep track of player's turns.

Player Turns

How do the methods called from `advance` work. In particular, how can we figure out if the current player moved and whether or not they won? `Moved` is pretty simple: the player moved if they moved to any of the tiles on the board.

```

245     isPlayersMove(t13) || isPlayersMove(t21) || isPlayersMove(t22) ||
246     isPlayersMove(t23) || isPlayersMove(t31) || isPlayersMove(t32) ||
247     isPlayersMove(t33);
248 }
249
250 /**

```

Listing 6.9: TicTacToe `isPlayer Moved`

`isPlayer Moved` is an example of a very long (though not very complicated) Boolean expression. Remember that logical `or` is true if either or both of the expressions it joins are true. Thus this long `or` is true if any of its subexpressions are true. What is `isPlayersMove`?

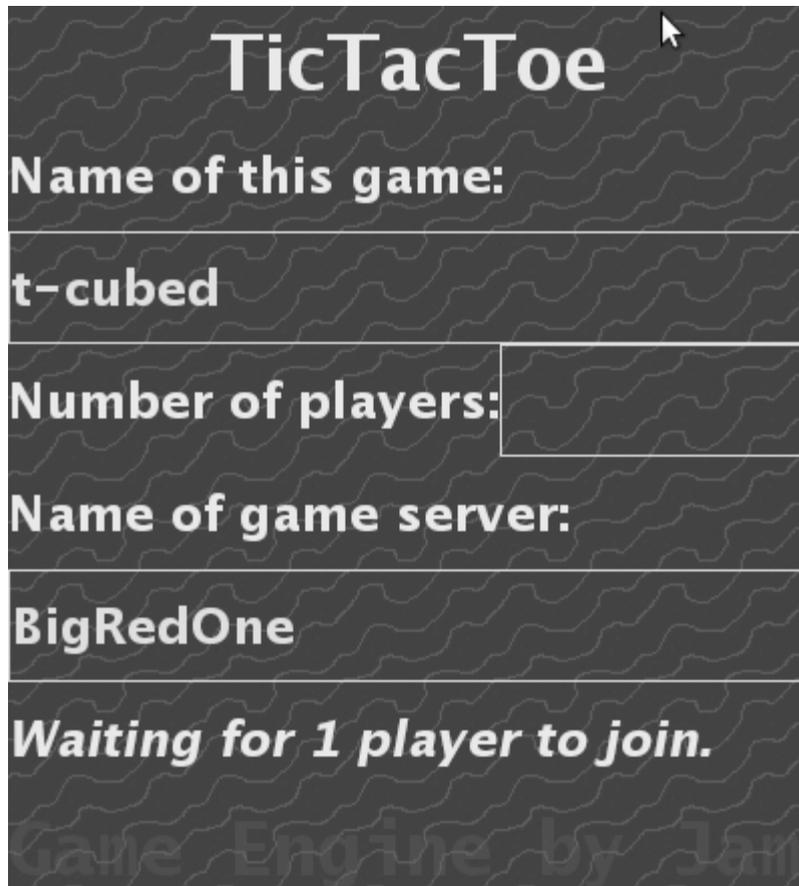


Figure 6.3: Waiting for a second TicTacToe player

```

107     gt.setContent(currentPlayerSymbol);
108     ++turnsTaken;
109     return true;
110   } else {
111     return false;
112   }
113 }
114
115 /**

```

Listing 6.10: TicTacToe isPlayersMove

`isPlayersMove` takes a `GameTile` as its parameter and it determines if the player *can* move there (it must currently be empty) and whether the player *did* move there (was the tile clicked by the current player). If they can and did, then put their symbol in the tile and add one to the move count.

```

224     col1(player) || col2(player) || col3(player) || diag1(player) ||
225     diag2(player);
226   }
227
228 /**

```

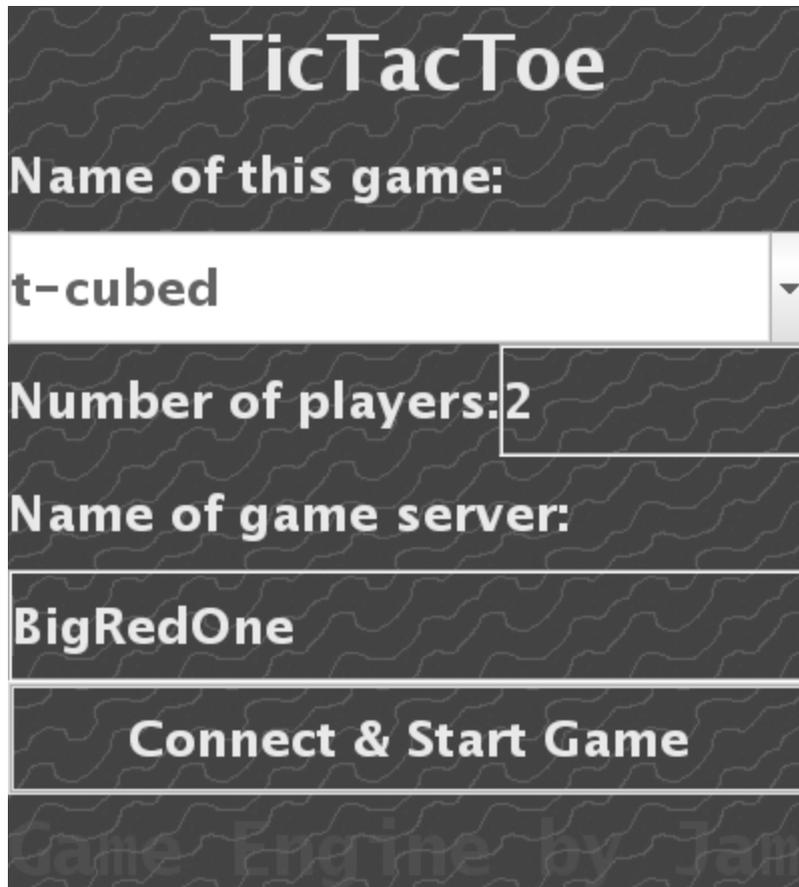


Figure 6.4: Joining “t-cubed” TicTacToe session

Listing 6.11: TicTacToe isWinner

The `isWinner` method is another long or. This time, however, it calls eight *different* methods. There are eight methods because there are eight different ways to win: 3 rows, 3 columns, and 2 different diagonals. We will just look at one of the methods as the other seven are almost identical; it would be nice to have some way to avoid repeating ourselves here.

```

124     t13.getContent().equals(p);
125 }
126
127 /**

```

Listing 6.12: TicTacToe row1

`row1` (and all the other individual win checkers) takes a player symbol, a `String` as its sole parameter. That is possible because only the player who just made a move could possibly have won. So we don’t have to call `isWinner` with both symbols. It would not be that much harder to call it twice (or once for each player). Knowing what player we are checking simplifies the individual win checking methods.

Comparing objects. Note the use of the `equals` method to compare the contents of a `GameTile` with the player’s symbol. This method is defined in the base class of *all* Java classes, `Object` and is overridden in objects

which need a special way to compare themselves for equality. The `equals` method takes an `Object` as its parameter and compares the given object to `this`. For a `String`, `equals` returns `true` if the two strings have exactly the same number of characters and both strings match in each character position.

What about `==`? That is what we used to compare `int` and `double` values. Why not use it for `Object` and its child classes? Java *permits* the comparison of `Object`-derived classes with double equals; comparing them with `==` has a different meaning than what we normally mean.

If we create two new `String` objects, each with the same contents:

```
String one = new String("Hello, World!");
String two = new String("Hello, World!");
boolean doubleEquals = (one == two);
boolean equalsMethod = (one.equals(two));
```

What are the values of `doubleEquals` and `equalsMethod`? the first Boolean expression, using `==` returns `true` if and only if the two references refer to the same object; it returns `false` otherwise. The `equals` method, on the other hand, does whatever `String` defines to be the comparison operation for objects of its type (this is an example of encapsulation in a type; the meaning of “equality” is left to the implementer of the class).

Thus the two Boolean expressions return `false` and `true`, respectively. This is the circumstance on the left-hand side of Figure 6.5.

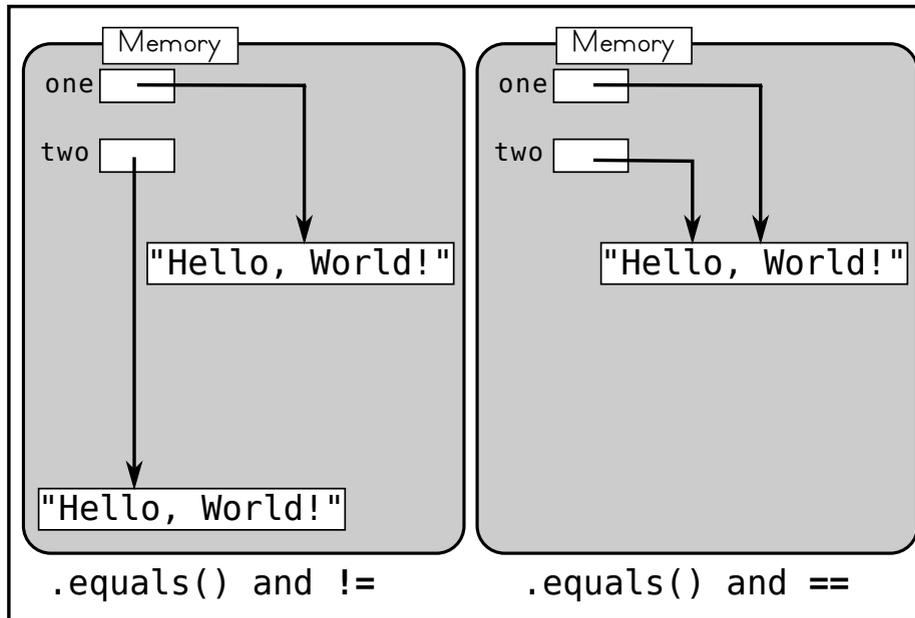


Figure 6.5: Java references, `==` and `equals()`

On the right-hand side of the figure is the result of the following code snippet:

```
String one = new String("Hello, World!");
String two = one;
boolean doubleEquals = (one == two);
boolean equalsMethod = (one.equals(two));
```

The only difference is that the value assigned to `two` is *not* a newly created string with the sequence of characters “Hello, World!”, but rather a reference to the already created string containing those characters. Since the sequence of characters in `one` and `two` (or rather, the actual strings to which they refer) is the same, this code sets `equalsMethod` to `true`; since the values *inside* the references are the same, `doubleEquals` is also set to `true`. Now back to our regularly scheduled program.

The `isCatsGame` method is very simple: it checks if the number of moves is 9. If 9 moves have been made, there are no empty squares. If we already know there is no winner from the last move, the game must end in a tie.

The `handle...` methods are also pretty simple: they determine a message to display and call `endGame` with the text of the message to display.



Figure 6.6: X wins TicTacToe

```

245     isPlayersMove(t13) || isPlayersMove(t21) || isPlayersMove(t22) ||
246     isPlayersMove(t23) || isPlayersMove(t31) || isPlayersMove(t32) ||
247     isPlayersMove(t33);
248 }
249
250 /**
251  * Create a StringSprite with the given message and display it across
252  * the screen; also make sure that the game is over.
253  *
254  * @param announcement the end-of-game announcement (winner or

```

Listing 6.13: TicTacToe endGame

The `endGame` method builds a `StringSprite` with the message (who won or that it is a tie) and puts it on the screen as you can see in the image above. It also calls `setGameOver`, a method in `Game`. That means advance will do no more work (look at line 294; `isGameOver` will return `true` after it is set).

Both games in the session see the exact same sequence of input. When either player moves or clicks their mouse, the information goes to both versions of the game. The only clicks that make a difference, though, are those by the current player on empty squares (`isPlayer sMove` makes sure the square is empty and `isPressed` in `GameTile` takes a player number to make sure only that player's clicks count). Even though two “different” versions of the game are running, the input stream is shared so the two games unfold identically. The games you have built in previous chapters can also be extended for multiple players.

6.5 Summary

Network Communications

A running program or *process* can communicate with other running processes across the network. FANG, through Java, supports multi-player games where multiple players run the same game in multiple processes.

Network communications can be from *client* processes to *server* processes where the servers provide some special service to the clients. This architecture is often many clients to one server. An alternative architecture is for many *peers* to connect. Peers are all the same program running as separate processes.

FANG uses a client/server architecture. The server portion runs and all clients send their input to the server. The server then sorts all the input into a particular order and sends it to all of the clients. Clients only run the game with the input they get back from the server. This lets each program run the program from the beginning and, since they see the exact same sequence of input, the programs remain synchronized.

Abstraction

Abstraction is the encapsulation of rules and data together in a new type, an *abstract data type*. An abstract data type has three parts:

1. A *public interface*. The collection of public methods.
2. A *private implementation*. Also known as a “layout in memory”. For classes this means the private fields and methods and internal types.
3. Interaction is limited to the public interface.

Abstraction permits a part of the solution to be wrapped up inside a `class` so that users can focus solely on the interface while the implementer can ignore how the type is being used.

FANG Classes and Methods

Java Templates

```
public class <ClassName>
  extends <ParentClassName> {
  <static definitions>
  <field definitions>
  <constructor definitions>
  <method definitions>
}
```

Chapter Review Exercises

Review Exercise 6.1 If there were three players in a networked game, how would you check to see if the first player had typed 'y'?

Review Exercise 6.2 What are the advantages of sharing just the input stream in a FANG game? What risks are there?

- (a) If I told you I was running TicTacToe on my computer, nonesuch.potsdam.edu and I had started session t-cubed, describe the steps you would take to connect to my session.

Review Exercise 6.3 Where do you specify the number of players necessary to start a multi-player game?

Review Exercise 6.4 What is a `static final` field for? What advantages are there to using named constants?

Review Exercise 6.5 Why do you think using `final` whenever possible is a good idea? Or, if you want to argue the contrary, why is it *not* a good idea? Support your position with consideration of how easy/difficult your code is to understand, how long your code is, and how much effort the code is to change.

Review Exercise 6.6 Why do we use setter methods rather than making fields `public`?

Review Exercise 6.7 If `String` (`java.lang.String`) is *immutable*, what can you tell me about *all* of its fields.

Review Exercise 6.8 When using a `static` method in `RectangleSprite`, what comes before the dot?

Review Exercise 6.9 What does the annotation `@Override` before a method mean? What checks does the compiler perform for you when it is there? What does it tell the programmer?

Review Exercise 6.10 How would you let *either* player in `TicTacToe` pick the next square to move to? (Note: This makes for a mildly interesting “click as fast as you can” sort of game; it is *not* a good game.)

Review Exercise 6.11 What is the advantage of making as many methods `private` as possible? These methods are or are not part of the public interface of the class? If not, what *are* they a part of?

Programming Problems

Programming Problem 6.1 Start with `TicTacToe.java`. Modify the program to play a 4 × 4 game (with 4 in a row required to win).

- Does it seem likely that `GameTile` will require changing?
- Where will you declare new `GameTile` fields?
- Make sure to modify `makeGameTile` so it sets the right sizes. Modify `setup` to construct all 16 tiles.
- How many ways are there to win in the new game? Write the required helper functions and rewrite `isWinner` accordingly.
- Is there anything else that must be changed?

Programming Problem 6.2 Start with `TicTacToe.java`. How would you modify the program so that it can be played in a single FANG instance with the players passing the mouse back and forth? What is the *minimum* changing you can do?

Programming Problem 6.3 Start with `SoloPong.java`. Modify `SoloPong` so that it plays a two player version of the game. Add a second paddle. When constructing a paddle you will probably want to take a player number as a parameter and keep it as a field in the paddle so that each paddle responds to just one of the players.

Playtesting. You will want to play the game some and then consider adjusting how the ball starts. You will want to move the location so it is closer to the center and be able to vary the direction according to which player last missed the ball. You will want to keep score for each player individually.

Programming Problem 6.4 One problem with `TicTacToe` is that it plays a single game and then must be restarted to play a new game. Can you modify the game so that it advances to a new game when the current

game is over? Or better, that pauses for 10 seconds so players can see the results and then starts the game over.

You will want to switch which player is X and which is O. What changes when player 1 is X and player 0 is O?

Programming Problem 6.5 Look at Problem 2. Modify the two player version of `EasyDice` to make it a networked, multi-player game. If you have a one-machine solution, the real multi-player solution should be very simple (though `EasyButton` must be modified to make it like `GameTile`).

Programming Problem 6.6 Look at Problem 3 for a description of the game of *Pig*. Using `OneDie` and, perhaps, `EasyDice`, you can build a multi-player game of *Pig*. Start with making it two player; after the next chapter you should be able to extend it to an arbitrary number of players.

Collections: ArrayLists and Iteration

When writing `TicTacToe`, much of the length of the program was taken up with methods to test for winning combinations. If you look at several of them, say the `row*` methods, one very interesting thing to note is how similar the three methods are.

```
118     * @param p player to check
119     *
120     * @return true if p won across row 1; false otherwise
121     */
129     *
130     * @param p player to check
131     *
132     * @return true if p won across row 2; false otherwise
140     * Did the given player (X, 0) win across row 3?
141     *
142     * @param p player to check
143     *
```

Listing 7.1: TicTacToe the row winning methods

Note that all three use the same form, a logical or between three calls to the `equals` method. `equals` is the standard way of checking whether or not two objects are equal. What is interesting is the similarity of the names of the variables at the beginning of each chained call to `equals`.

Notice that the three methods are actually identical except for the names of the variables: `row1` uses field names beginning `t1`; `row2` uses field names beginning `t2`; `row3` uses field names beginning `t3`. Thinking about our desire to avoid repeating ourselves, this seems wasteful. Further, imagine the changes necessary to extend the game to play 4 4 tic-tac-toe. More methods with more similar variables in each method.

There has to be a way to get Java to do the work, to have it calculate which variable we want to address at run-time, saving the programmer work and headache at compile time.

This chapter is about *collections*, objects which contain other objects. A collection has some method of indexing to recover individual objects in the collection. We will start with collections indexed by small integers so that we can *loop* across all entries in the collection, executing the same code on each entry.

7.1 Flu Pandemic Simulator

This is a book about games. Why is there a section titled “Flu Pandemic Simulator”? The topic is certainly too dark for a book on simple computer games and it is surely too complex for a beginning programming book.

Actually, neither of the statements in the preceding paragraph is true. While a pandemic simulator may well be dark, the idea of simulating the real world, using numbers taken from real pandemics of the future, can give users of the program some feeling for how dangerous such a situation could be. And it gives them that insight *before* anyone dies. Also, by adding some game elements, with the player directing limited protective resources, players can understand why proactive preventive measures are a good idea. It also gives the author a good chance to introduce the ideas of serious games.

A pandemic simulator is also not too difficult to program. The world, in such a simulation, consists of people in a village. The villagers are modeled as simply as possible: each maintains its own health and its own visual representation. Age, gender, hair color, name, all might be interesting features of a real person but the point of a computer model is to simplify as much as possible. All we are interested in is the progression of a single illness; people thus become what state their health is in and how many days they have been in that condition. For example, the simplest model would be HEALTHY and SICK as the states of health. Healthy people do not become sick spontaneously (unless exposed to the sick) and sick people become well after three days in bed. This description, in some way, matches the common cold. The only model parameters missing are the mortality rate (percentage of people killed by the disease), the infection rate (percentage of people exposed to the illness who become ill), and some way of modeling the relationships within the village.

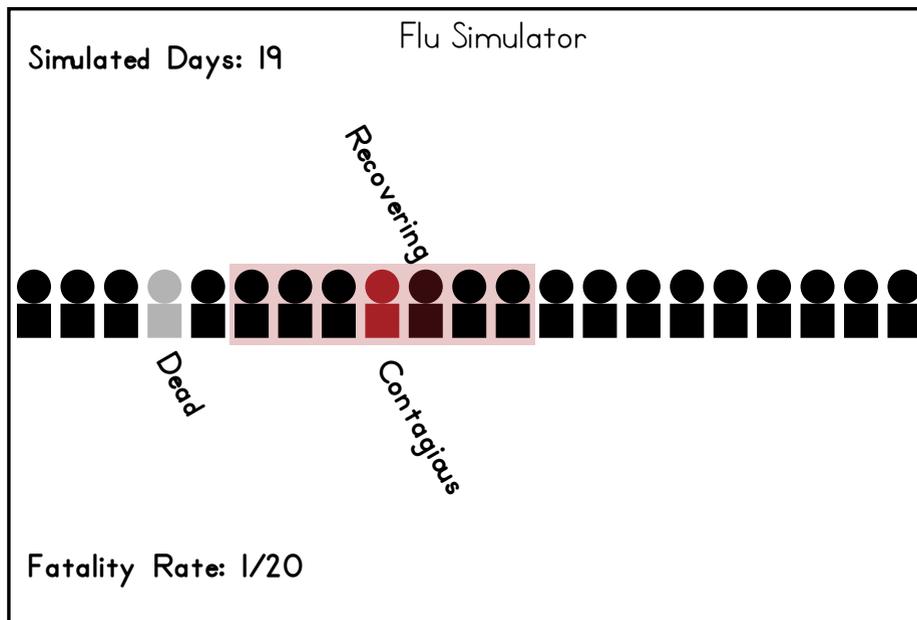


Figure 7.1: FluSimulator game design diagram

The *FluSimulator* will represent the village in the middle of the screen as a row of people. Each person will change color to reflect their current state of health. The design also introduces a more complex model of disease: an infected individual is sick but not contagious for some amount of incubation time; then, for some amount of time, they are contagious; then, for another amount of time, they are sick but recovering. Both healthy and dead individuals do not change their state spontaneously. This means there are five states of health (and five colors in the final game): DEAD, HEALTHY, SICK, CONTAGIOUS, and RECOVERING.

Introduction to Discrete Time Simulation

Having conceptualized a simplification of a person as their state of health, how will we model a pandemic. The missing dimension of our simulation is time. Like movement, in the real world, time unfolds continuously; also like movement, when we model the time dimension, we will have to move forward in discrete steps. What that means is that we will set an arbitrary period, say one day, and we will apply all changes to the village at one day intervals.

Why choose a day? Because that seems like a reasonable level when looking at a disease which lasts about a week. If we were modeling subatomic particles, it would make sense to use a much finer time step. One reason for using discrete time steps is that continuous time requires expressing the relationships between time and various persons in the village as linear differential equations, a complicated process. If we step time at discrete intervals, it is much easier to describe the relationship between the states of health.

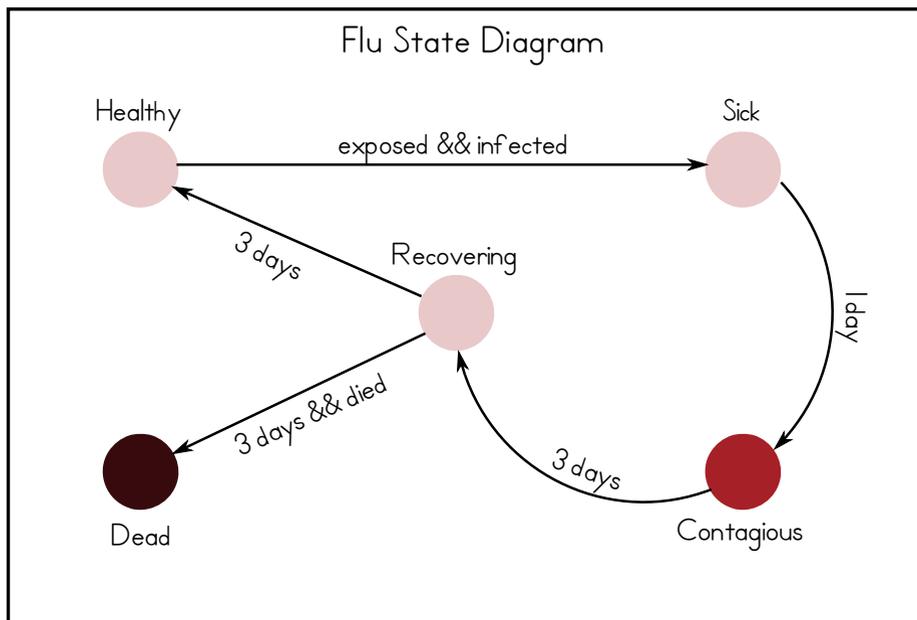


Figure 7.2: Flu Health State Diagram

A *state diagram* like that in Figure 7.2 can be used to describe the states of an object and how that object can change states. In this diagram the circles represent a person's state of health. Each is labeled with the name of their health state. The arrows indicate what *transitions* are permitted between states. In our simple disease model, there is no way for a HEALTHY¹ person to die (go directly to DEAD). Each transition arrow is labeled with the event which causes the person to make that transition. Thus a person must be exposed to the disease *and* become infected with the disease to go from HEALTHY to SICK.

The two things to note about the state diagram are that there are two transitions out of RECOVERING and there are *no* transitions out of DEAD. The decision between the two ways out of RECOVERING is made by consulting the *mortality rate* of our disease. After three days pass, if our person is found to be in those killed by the disease, they die. If not, then they become healthy again. The decision of mortality (as well as exposure and infection) will use random numbers.

This model is very simple. At the end of each day a person can check their current health state and the number of days they have been in the current state. With that information, the person can determine whether or not they should transition to a new health state. If the person is leaving RECOVERING, they will also need to determine whether they live or die.

¹We will use the named constants from the program, all capitalized and in the standard code font. The state labels in the diagram use mixed case as it reads much better in the diagram font.

Note that upon becoming healthy again, a person can be reexposed and reinfected. This does not match disease spread exactly but for the simplicity of the model we will leave it (giving a person a “memory” of being sick is discussed in Programming Problem??).

Defining Neighbors and Chance of Infection

How is a person exposed? Once a person is exposed, how is it determined if they are infected? Look back at Figure 7.1; notice that the villagers are all in a row. That is, for our model, assumed to represent the main street through the village. Thus proximity along the line equates to proximity in the village. We use a simple social model, one where people, sick or well, are likely to visit their “neighbors”. Neighbors are those villagers within some fixed distance to the left and the same fixed distance to the right of the person. Thus with a 3 person neighborhood, the CONTAGIOUS person’s neighborhood is represented by the pink rectangle.

In our social model, each person interacts with a fixed number of neighbors per day (any number of which may, in fact, be the same neighbor or even themselves). If the person happens to be CONTAGIOUS when interacting with a neighbor, then the neighbor is *exposed* to the disease (the first part of the transition from HEALTHY to SICK). When a person is exposed, the *virulence* or contagiousness of the disease is consulted and if the exposed person was HEALTHY and a random number says they were infected, they become SICK.

The social model is actually just a little bit simpler: we do not keep track of everyone’s interactions with their neighbors, just those interactions coming from CONTAGIOUS persons. This means that we don’t check to see if the neighbors of the CONTAGIOUS interact with the CONTAGIOUS on any given day. This means the infection rates are lower than they would be in the model described above. Programming Project ?? discusses how to extend the program with a symmetric social model.

Putting a Game in the Simulation

As described, the above is a *simulation* rather than a game. We have components (villagers) and we have rules (both a disease model and a social model), so what is missing? The “player” has no decisions to make and therefore does not influence the outcome of the simulation. The player is not *playing*.

There are several ways to extend this into a game: let the player move people around on the screen to change the social model and exposure; give the player a limited number of vaccines which make people immune; give the player a limited amount of medicine which increases recovery or lowers mortality or infection rates. Given the new material necessary to handle having 20 villagers in the village, these extensions into a game will be left as programming exercises.

This chapter continues with an introduction to *console* programming. This permits output to the console window (we will write whole programs in setup, kind of like we did in Chapter 2). Then it introduces the idea of collections, a type of object which can contain a collection of other objects (for a village full of people). Then it returns to the simulation, describing the Person class’s public interface and using the collection techniques to advance the simulation, determine when the simulation is over, and to determine the overall mortality rate of the disease.

7.2 Console I/O: The System Object

Java has a System object. The full name of the System type is `java.lang.System`; any types defined in the `java.lang` package² do not need an `import`; they are all imported automatically by Java.

Looking System up in the JavaDoc documentation, one finds that its public interface consists solely of **static** methods and three **public, static fields**. Yes, this goes explicitly against the admonition to never have **public** fields. That is all we will say about that and we will use one of those fields, `out`, to generate console output.

²The fully qualified names of classes are some collection of *packages*, separated by dots, and then, at the end, the class name. If no package names are specified, it means that the class is in the current directory; classes defined in the same directory do not need to be `imported` at all.

Using the Console for Output: `System.out`

Take another look at the `System` documentation. Notice that the type of the field `out` is `PrintStream`. Also notice that the name of the type is, in fact, a link to the documentation for that type. The JavaDoc document compiler is aware of the Java language and attempts to link all the types to their documentation. Follow the link to see the public interface for `PrintStream`.

The `PrintStream` class has two methods that interest us at the moment: `print` and `println`. Both of these methods come in a great many flavors, all with a single parameter (each with a different type so the compiler can tell them apart). Their purpose is to print the value of the expression passed in to the console. To show this in action, we will write `HelloWorld.java` in FANG:

```

1 import fang.core.Game;
2
3 /**
4  * This "game" has no sprites. In setup it just prints "Hello, World!"
5  * to the standard output console. This is a demonstration of how
6  * System.out can be used.
7  */
8 public class HelloWorld
9     extends Game {
10    /**
11     * Print out one line. That is all the setup required
12     */
13    @Override
14    public void setup() {
15        System.out.println("Hello, World!");
16    }
17 }

```

Listing 7.2: `HelloWorld` Demonstrate `System.out`

When the program is run, it looks exactly like `EmptyGame.java` (except for the name in the title bar). The interesting thing is what happens in the console window where you ran the program³.

Figure 7.3 shows the command-line window where the program was run after being compiled. Note that the `classpath` given in the screenshot is that on the author's main machine and probably doesn't match your class path.

The only difference between `println` (as used in this program) and `print` is that `println` prints an end-of-line character after printing the parameter. Using `print` it is possible to build up a single line of output over many calls to `print`; this is useful when printing while using *iteration*; you might only want to start a new line at the end of the iteration. More on this below when we try our hand at iteration.

Poor-man's Debugger

What good does console output do us? It can be used to give the user feedback that does not fit well into the game display. Because `print` and `println` can print many different types, it is often easy to add lines that print out the current values of a variable at several points. Consider the output of `PrintASprite.java`.

```

1 import fang.core.Game;
2 import fang.sprites.RectangleSprite;
3
4 /**
5  * Demonstrate how a Sprite class object prints out using
6  * System.out.println.

```

³If you are programming in a browser using `JavaWIDE` or another on-line IDE, you will need to open your Java Console to see console output. Exactly how to do that is browser specific.



```
File Edit View Terminal Tabs Help
~/Chapter07% java -classpath ../../../../vendor/jars/fang.jar HelloWorld
Hello, World!
```

Figure 7.3: Console output of HelloWorld

```

7  */
8  public class PrintASprite
9  extends Game {
10  /** The value to print out. */
11  private RectangleSprite someField;
12
13  /**
14   * Creates a rectangle sprite for the field; print field value several
15   * times to show toString method
16   */
17  @Override
18  public void setup() {
19      System.out.println("someField = " + someField);
20      someField = new RectangleSprite(0.25, 0.25);
21      System.out.println("someField = " + someField);
22      someField.setLocation(0.5, 0.5);
23      addSprite(someField);
24      System.out.println("someField: " + someField.getX() + ", " +
25          someField.getY() + " - " + someField.getWidth() + ", " +
26          someField.getHeight());
27  }
28  }

```

Listing 7.3: PrintASprite Demonstrate System.out

The game declares a `RectangleSprite` field (line 13). The `setup` method runs sequentially. Thus the field is printed twice, once before and once after the call to `new`. This book has been adamant that you not use fields before they are initialized; if you look at the first line of output below, you will see why:

```
~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar PrintASprite
someField = null
someField = fang.sprites.RectangleSprite[x = 0.0, y = 0.0],
[w = 0.25, h = 0.25] color = FANG Blue
someField: 0.5, 0.5 - 0.25, 0.25
```

The value of the uninitialized field is `null`; `null` is the reference value that means the reference refers to *nothing*. The important thing is that you cannot apply the dot operator (.) to the `null` pointer without causing a run-time error.

The next line in the output is the `String` value of the `RectangleSprite` right *after* the call to `new`. In Java all object types have a `toString` method which is used to convert a non-null reference to an object into a string. This means `toString` is part of the *inherited* public interface of every class; inherited because it comes from extending an object which has the method in its public interface.

The implementation of `Sprite` overrides the definition of `toString` it inherits with the following code:

```
/**
 * Return a string representation of the {@code Sprite}. The type,
 * location, size, and color of the {@code Sprite}.
 */
@Override
public String toString() {
    return getClass().getName() + "[x = " + getX() + ", y = " + getY() +
        ", " + "[w = " + getWidth() + ", h = " + getHeight() +
        "] color = " + Palette.getColorName(getColor());
}
```

When an object type is passed to `printf` or, in this case, when it is passed as a parameter to the `String` version of the `+` operator, Java converts the value to a `String` with a call to the `toString` method of the object. In Java, when a method is overridden, the compiler calls the version defined farthest down the extend hierarchy for the *constructed object type*. Here, we can see the call to `new` in line 22 and see that the object is a `RectangleSprite`. Looking at the documentation page for `RectangleSprite`, we see the hierarchy of extend:

```
java.lang.Object
+- extended by fang.core.Sprite
   +- extended by fang.sprites.RectangleSprite
```

Thus the version of any overridden method called will be: the one defined in `RectangleSprite`, if there is one; or else the one defined in `Sprite`, if there is one; or else the one defined in `Object`, if there is one; or else throw a compiler error. So, `toString` is defined in `Object`, a class which *every single object class in Java* extends. It is overridden in `Sprite`. It is *not* overridden in `RectangleSprite` (you can see if it is overridden in the documentation page: if the method is listed in the **Method Summary** portion of the page, it is overridden in that class; if, instead, the name of the method appears below the **Method Summary** in a box with a label like **Methods inherited from fang.core.Sprite**, then the method was last defined in the hierarchy of extend in the named class.

Since `Sprite` was the last class to define `toString` in the hierarchy of `RectangleSprite`, it follows that the `toString` called to create the second output line is the one defined in `Sprite`. We will discuss this selection of the lowest override in the *actual type's* hierarchy in more detail later.

The final line of output is generated by the `println` beginning on line 26. Remember that `+` used with `String` expressions concatenates them together. Here the first item in the `println` is a `String` literal. The remaining pieces are calls to methods on `someField` and punctuation strings to make the result readable. The final line gives the location and size of the sprite at the end of setup.

There is a program (included in most modern integrated development environments (IDE)) called a *debugger*. A debugger lets you pause a program at an given line and examine the value of variables, look at field values inside of objects, and even stop the program when the value of a given variable changes. There are tutorials on the Web on how to use debuggers in various IDEs; that instruction is beyond the scope of this book.

This use of `printf` is a way of doing much the same thing. At various points inside your programs, where you wonder what is happening to the location of a given sprite or something similar, you can add a `print` statement and track the value. This is useful even with an integrated debugger in that while a debugger lets you pause the program on each line, stepping the debugger through all of the start-up code of a FANG program (or any other framework program) can be tedious.

7.3 Iteration

How could you print the numbers 0 through 9 to the console as part of setup? Hopefully something like this occurred to you:

```
public void setup() {
    System.out.print(" " + 0);
    System.out.print(" " + 1);
    System.out.print(" " + 2);
    System.out.print(" " + 3);
    System.out.print(" " + 4);
    System.out.print(" " + 5);
    System.out.print(" " + 6);
    System.out.print(" " + 7);
    System.out.print(" " + 8);
    System.out.print(" " + 9);
    System.out.println(); // start new line
}
```

Two things to notice about the code: each line looks just the same with a single character different (seems to violate DRY) and it would be very hard to extend to 100 numbers (and harder for 1000). We need some way to *iterate* or do the same thing (or almost the same thing) over and over again.

The following code does the trick:

```
1 public void setup() {
2     for (int i = 0; i != 10; ++i) {
3         System.out.print(" " + i);
4     }
5     System.out.println(); // start new line
6 }
```

The *semantics* or the meaning of this language construct is based on the three parts of the **for** statement that are inside the parentheses. The three parts, separated by the semicolons, are described in the Java template for the **for** statement:

```
<forStatement> ::= for (<init>; <continuation>; <nextStep>) {
                    <body>
                    }
```

The *<body>* of the **for** loop is the collection of statements which are *iterated* or done many times. The *<init>* or *initialization* part of the **for** statement is executed *once* before any other parts of the statement are evaluated or executed. In the counting loop above, the *<init>* is **int i = 0**: this declares a local variable and sets the value of the variable to zero. Note that it is not *necessary* to declare a *loop-control variable* in the *<init>* though it is often convenient.

Question: What is the *scope* of *i*? The scope is the whole **for** statement. That is, *i* is visible inside all three parts of the **for** statement line *and* throughout the *<body>*. After initializing, execution of the **for** loop continues by evaluating *<continuation>*. The *<continuation>* portion is a Boolean expression. When *<continuation>* evaluates to **true**, the *<body>* is executed once; when it is **false**, execution continues *after* the body of the **for** statement. Thus in our sample loop above, when *i* is not *exactly* equal to 10, the body of the loop is executed. As you can see, right after initialization, **i != 10** is **true** so the body is executed.

Each time the statement executes the *<body>*, upon reaching the end, execution continues by executing the *<nextStep>* and then reevaluating the *<continuation>* condition. Again, if the Boolean expression evaluates to **true**, the execution does the body of the loop and then does the *<nextStep>* and *<continuation>* again.

So, what does our example loop do, exactly? I will unroll the loop, printing each of the three parts separately as they appear. This is not real Java in that it uses a special “statement”, EXIT to indicated that execution should move to the statement after the loop. So **if (boolean) EXIT;** means to evaluate the expression and if it is **true**, get out of the loop. Here is a trace of the execution of the various parts:

```

int i = 0;           // init: i = 0
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 0"
++i;                 // nextStep: i = 1
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 1"
++i;                 // nextStep: i = 2
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 2"
++i;                 // nextStep: i = 3
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 3"
++i;                 // nextStep: i = 4
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 4"
++i;                 // nextStep: i = 5
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 5"
++i;                 // nextStep: i = 6
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 6"
++i;                 // nextStep: i = 7
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 7"
++i;                 // nextStep: i = 8
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 8"
++i;                 // nextStep: i = 9
if (!(i != 10)) EXIT; // continuation
System.out.printf(" " + i); // body: " 9"
++i;                 // nextStep: i = 10
if (!(i != 10)) EXIT; // continuation
EXIT;
System.out.println(); // After loop

```

Note that *<init>* is executed *once* and that *<continuation>* is evaluated one more time than the body of the loop (it must evaluate to **false** once) and that *nextStep* is executed once at the end of each time the body is executed.

This is, in many ways, similar to *setup* and *advance*: *setup* is run once *before* the main video game loop is entered. After the game is entered, the body of the main video game loop includes a call to *advance*.

Practice Loops

This section will describe several sequences of numbers to print out. It will then present the **for** loop which would print out the numbers. At the end of the section a couple of points the loops have in common will be pointed out.

More Numbers

How would you change the example loop given above to print the numbers 0 through 99? We want 100 different numbers.

```

for (int i = 0; i != 100; ++i) {

```

```

    System.out.print(" " + i);
}
System.out.println(); // start new line

```

Clearly the change was just in the *<continuation>*: the 10 became 100. This is right but not aesthetically pleasing: there are too many numbers on one line. What if we want to print 7 numbers per line (with no more than 7 on the last line since 7 does not divide 100 evenly)? We need to check, inside the loop, whether we need to insert a new line. We will start a new line whenever *i* is a multiple of 7: a line before 0, 7, 14, 21,.... Note that this means there is a new line started before the first number is printed. We will look at how to avoid that after we get this loop working.

Also, note that the *<body>* of the loop is a collection of statements; it can contain calls to methods, **if** statements, or even more **for** loops.

```

for (int i = 0; i != 100; ++i) {
    if (i % 7 == 0) { // remainder of 0
        System.out.println(); // start new line
    }
    System.out.print(" " + i);
}
System.out.println(); // start new line

```

Here we test if *i* is a multiple of 7 by using the *modulus* operator, `%`. The modulus operator takes two **int** expressions and divides the first by the second, returning the *remainder* in the division. The definition of a number being a multiple of another number is that it have a remainder of 0 when divided by the number. Thus the Boolean expression in the **if** statement is **true** exactly when *i* is a multiple of 7.

To get rid of the first newline, it suffices to test if *i* is *not* zero as well as the multiple of 7 test:

```

15     System.out.println();
16     }
17     System.out.print(" " + i);
18     }
19     System.out.println();
20     }
21     }

```

Listing 7.4: MoreNumbers iteration

The output of this loop is:

```

~/Chapter 07% java -classpath ./usr/lib/jvm/fang.jar MoreNumbers
0 1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31 32 33 34
35 36 37 38 39 40 41
42 43 44 45 46 47 48
49 50 51 52 53 54 55
56 57 58 59 60 61 62
63 64 65 66 67 68 69
70 71 72 73 74 75 76
77 78 79 80 81 82 83
84 85 86 87 88 89 90
91 92 93 94 95 96 97
98 99

```

The columns do not line up (single digit numbers print in a single space while double digit numbers use two) but there are seven elements per line.

What if we want to do something other than print out *i*? What would we want to do? Perhaps add several randomly placed `RectangleSprites`?

```

16     curr.setLocation(randomDouble(), randomDouble());
17     curr.setColor(randomColor());
18     addSprite(curr);
19 }
20 }
21 }
```

Listing 7.5: IteratedRectangleSprites iteration

Figure 7.4 shows an example run of this program. This example uses iteration but not console output; these two techniques are independent of one another.



Figure 7.4: A Sample Run of IteratedRectangleSprites

Not Starting at 0

How would you print (on the console) the numbers from 10 up to 99? The easiest way is to change `<init>`: rather than set *i* to 0, set it to 10.

```

15     System.out.println();
16 }
17     System.out.print(" " + i);
18 }
```

```

19     System.out.println();
20     }
21 }

```

Listing 7.6: NotFromOne iteration

The output of this program is

```

~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar NotFromOne
10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31 32 33 34
35 36 37 38 39 40 41
42 43 44 45 46 47 48
49 50 51 52 53 54 55
56 57 58 59 60 61 62
63 64 65 66 67 68 69
70 71 72 73 74 75 76
77 78 79 80 81 82 83
84 85 86 87 88 89 90
91 92 93 94 95 96 97
98 99

```

Again, this is correct (the numbers printed are 10-99, inclusive) but it is not aesthetically pleasing. It would look better to have 7 elements on the first line and every line after except, possibly, for the last (how many numbers are printed? Is that number a multiple of 7? If not, how many elements will there be on the last line?).

The trick here is that the value *i* in previous examples has served two purposes: it contains the value to print next *and* it counts how many items have been printed. That second function must be done some other way now. How can we determine how many elements have been printed so far? Having that number, we can check if it is a multiple of 7 and start a new line.

When *i* is 10, 0 elements have been printed; when it is 11, 1 element; when it is 12, 2 elements. The pattern, so far, looks like the expression $i - 10$ tracks the number of numbers printed so far. Thus we can replace *i* in the **if** statement with $i - 10$:

```

15     System.out.println();
16     }
17     System.out.print(" " + i);
18     }
19     System.out.println();
20     }
21 }

```

Listing 7.7: NotFromOneFixed iteration

The parentheses make sure that the value of *i* minus ten is calculated before modulus or comparison. The output is:

```

~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar NotFromOneFixed
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31 32 33 34 35 36 37
38 39 40 41 42 43 44
45 46 47 48 49 50 51

```

```

52 53 54 55 56 57 58
59 60 61 62 63 64 65
66 67 68 69 70 71 72
73 74 75 76 77 78 79
80 81 82 83 84 85 86
87 88 89 90 91 92 93
94 95 96 97 98 99

```

Not Counting by 1s

How would we change the original loop to print out the first ten *non-negative multiples of 7*? You might think of modulus with multiples of 7 but this problem is simpler: since 0 is a multiple of seven, each larger multiple of seven is seven higher than the previous multiple of seven. That is, the beginning of the sequence is 0, 7, 14, 21,... How would we only print the multiples of seven? Could use the `if` statement from `MoreNumbers` but it makes more sense to just have `i` only take on multiples of 7 values. Instead of counting by 1, count by 7. This means `<nextStep>` is modified to `i = i + 7`:

```

15     }
16     System.out.println();
17 }
18 }

```

Listing 7.8: CountBySeven iteration

The other change is the limit on the loop: the first ten non-negative multiples of seven are less than 70 and the eleventh is 70. The output of this loop is:

```

~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar CountBySeven
0 7 14 21 28 35 42 49 56 63

```

Never Execute the Body

Finally, what if continuation is `false` from the beginning? Consider

```

15     }
16     System.out.println();
17 }
18 }

```

Listing 7.9: NeverExecuteBody iteration

What is the output of this snippet of code? The variable is initially 0 which is *not* less than 0. Thus `<continuation>` is `false` from the start and the `<body>` (and `<nextStep>`) is never executed. The only output is a newline.

About Iteration

In Java, `for` is one of the main ways of implementing iteration. Most of our examples used `!=` in the `<continuation>`; this is idiomatic in the language if you know the exact value the loop control variable will take at the end of the iteration. When we count by 1, this is pretty simple. The instance of counting by 7 was harder since we had to figure what value the variable would take (it had to be a multiple of 7).

It is also idiomatic to start counting with 0. Computer scientists start counting with 0 because when there is a collection of objects (as we will see in the next section), they are indexed by integers starting with 0. The reason is that a collection can then be stored as the computer address of the whole collection and a given element is found by adding the index times the size of each object. This will be discussed again in the next chapter. Also, there are multiple ways to do many things in a loop. Rather than adding 7 in the loop, we could just count from [0-10) (the “[” means that end point is included; the “)” means that end point is not) and multiply the value of `i` by 7 when we go to print it. This version is left as an exercise for the reader.

7.4 Collections: One and Many

Can we combine what we just learned about iteration with the code from `TicTacToe`? That is, can we avoid repeating ourselves in three (or eight) different routines for checking for a winning row of symbols?

The answer is no: the names we give variables such as `t11` only exist at *compile-time*. There is no way to iterate over those names. Fortunately most programming languages support *collections* or *containers*, types which can contain multiple other values.

Dynamic and Static Information Attributes of programs, and, in particular, variables, which are known at *compile-time* are referred to as *static* attributes. Attributes of programs which cannot be known until *run-time* are referred to as *dynamic* attributes. Because static attributes are known early, they cannot change during the life of the program.

Static attributes include the value of literals (such as `1`, `8`, `true`, or `"Hello, World!"`), the *type* of a variable, or the *name* of a variable. Dynamic attributes include user input (such as the timing and location of a mouse click), the result of a call to `randomDouble()`, or the *value* of a variable. Dynamic attributes can typically change over time (though we make use of the `final` designation to indicate that the variable can only be assigned a value *once*) while static attributes cannot.

The discussion of dynamic and static attributes appears here because we will be discussing two types of collections, arrays and `ArrayLists`. Arrays are built-in types which can have as elements any one type of object; arrays are created with a call to `new` which includes the exact number of entries which the collection can contain. `ArrayList` objects are library types which can have as elements any *object* type; they are created with a call to `new` which need not include the number of entries they are expected to hold and their size can be grown to accommodate any arbitrary number of elements.

Java has a group of standard collection classes. We will focus on the use of the `java.util.ArrayList` class because it supports adding an arbitrary number of elements to it at run-time, random access of elements based on their insertion order, and the ability to loop across all of the elements it contains⁴

Declaring an ArrayList

An `ArrayList` is a standard class. It is defined in the `java.util` package. That means the required import is

```
import java.util.ArrayList;
```

To declare an `ArrayList` field, you specify the access level, the type, and the name just like any other field. Except that the type includes both `ArrayList` and the type of objects stored in the `ArrayList`. For example, we will write a program to store `Integer` objects in an `ArrayList`. As you can see, `Integer` is an object type (the capital letter); from the name you can deduce that it is *like* the plain old data type `int`. That intuitive deduction is sufficient for our purposes here. Further discussion of the `Integer` type will be deferred until the end of this section.

So, to declare a field, `theTable`, an `ArrayList` holding `Integer` values, we use the line:

```
private ArrayList<Integer> theTable;
```

Just as we printed a `RectangleSprite` to the console, we will start by printing an `ArrayList` using this program:

```
1 import fang.core.Game;
2
3 import java.util.ArrayList;
4
5 /**
6  * Demonstrate how an ArrayList prints using System.out.println.
7  */
```

⁴There is a built-in type, the array, which is similar except for the variable size; we will discuss arrays as we need to use them.

```

8 public class PrintAnArrayList
9     extends Game {
10    /**
11     * ArrayList of Integer (we will put in some numbers and do things
12     * with them)
13     */
14    private ArrayList<Integer> theTable;
15
16    /**
17     * Create the ArrayList; print out some information about it
18     */
19    @Override
20    public void setup() {
21        System.out.println("theTable = " + theTable);
22        theTable = new ArrayList<Integer>();
23        System.out.println("theTable = " + theTable);
24        System.out.println("theTable.size() = " + theTable.size());
25    }
26 }

```

Listing 7.10: PrintAnArrayList on System.out

Without doing more than declaring the field, constructing a new object for the reference, and then using one method found in the public interface described in the `ArrayList`, `size` which returns the number of entries in the collection. The output of this program is:

```

~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar PrintAnArrayList
theTable = null
theTable = []
theTable.size() = 0

```

An `ArrayList` is an object so the first printf, line 23, prints `null`. Then the collection is instantiated: notice that the constructor called with the `new` operator includes the name of the element type just like the declaration of the field. Line 25 prints out the value of the `ArrayList`. Looking at the documentation for `ArrayList`, the `toString` method is inherited from `class java.util.AbstractCollection`; the exact location of the method is not as important as our ability to find the documentation for the routine that is called (it helps us debug). According to the documentation:

`toString`

```
public String toString()
```

Returns a string representation of **this** collection. The string representation consists of a list of the collection's elements in the order they are returned by its iterator, enclosed in square brackets ("`[]`"). Adjacent elements are separated by the characters `,` (comma and space). Elements are converted to strings as by `String.valueOf(Object)`.

Overrides:

`toString` in class `Object`

Returns:

a string representation of this collection

Thus the two square brackets on the second line of output indicate that the `ArrayList` is empty. This is confirmed by the last line of console output where the size of the `ArrayList` is 0.

Filling a Collection

So, how do we put values into the collection? The `ArrayList` class defines a method, `add`. The `add` method comes in two flavors: with one parameter of the type of elements in the list, it adds a new element at the end of the `ArrayList`; with two parameters, one an `int` index and the other an element, it inserts the element at the given location, moving all the other elements up one index. The one parameter version is the more usual.

So, to put some multiples of 7 into the `ArrayList` we can use the following:

```

23     System.out.println("<before> theTable = " + theTable);
24     System.out.println("<before> theTable.size() = " + theTable.size());
25     theTable.add(63);
26     theTable.add(56);
27     theTable.add(49);
28     theTable.add(42);
29     System.out.println("<after> theTable = " + theTable);
30     System.out.println("<after> theTable.size() = " + theTable.size());
31 }
32 }
```

Listing 7.11: `PrintArrayList2` on `System.out`

The output generated by this version of the program is:

```

~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar PrintArrayList2
theTable = null
<before> theTable = []
<before> theTable.size() = 0
<after> theTable = [63, 56, 49, 42]
<after> theTable.size() = 4
```

After adding 4 elements, the 4 elements are in the `ArrayList` *in the order they were added*. This is similar to `addSprite`: with plain `add`, the newest item is “on top” in the sense that it is last in the `ArrayList`.

Now, what if we wanted the elements to be the first ten non-negative multiples of 7 and we wanted them in ascending order? That is, 0 should be first and 63 should be last.

Yes, we could, easily, cut and paste the four lines in the current version and then go through and modify each line and...we saw how to print out the numbers we want on the screen. Can we reuse that loop?

```

15     }
16     System.out.println();
17     }
18 }
```

Listing 7.12: `CountBySeven` iteration (again)

The body of the loop is a call to `print`. We would rather call `add` on our `ArrayList`. That becomes:

```

23     System.out.println("<before> theTable = " + theTable);
24     System.out.println("<before> theTable.size() = " + theTable.size());
25     for (int i = 0; i != 70; i = i + 7) {
26         theTable.add(i);
27     }
28     System.out.println("<after> theTable = " + theTable);
29     System.out.println("<after> theTable.size() = " + theTable.size());
30 }
31 }
```

Listing 7.13: `PrintArrayList3` iteration

```
~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar PrintAnArrayList3
theTable = null
<before> theTable = []
<before> theTable.size() = 0
<after> theTable = [0, 7, 14, 21, 28, 35, 42, 49, 56, 63]
<after> theTable.size() = 10
```

So far we know:

- An `ArrayList` is a generic object
- *Generic* means that the type name requires another type name between angle brackets to be complete *and* that type must be an object type. For example, `ArrayList<Integer>`.
- Somewhere above `ArrayList` `toString` is overridden to print out the contents of the list.
- The `add` method adds elements at the end of the list or at a given location (one or two parameter versions).

Iteration: Traversing a Collection

Given that we can insert elements into a collection, how can we get values out? Of course we know how to *print* the contents of the `ArrayList` on the console (using the overridden version of `toString`). Now what we want to do is access each element so that we can do something with it other than print. We want to get the value of each element so that we can treat the collection as a collection of *variables*. That is, if we added 9 `GameTile` objects to a properly declared `ArrayList` (what would the declaration look like?), we could examine each in the `ArrayList` to determine who had won the game of `TicTacToe`.

The method we need is called `get`. In an `ArrayList`, `get` takes an `int` index, the numeric position of the element in the array list and returns a reference to the element in the collection.

Manipulating Each Element

We can use the already seen `size` method along with our standard loop construct to print out all of the elements in an `ArrayList`:

```
23     for (int i = 0; i != 70; i = i + 7) {
24         theTable.add(i);
25     }
26
27     for (int j = 0; j != theTable.size(); ++j) {
28         System.out.print(" " + theTable.get(j));
29     }
30     System.out.println();
31 }
32 }
```

Listing 7.14: `PrintAnArrayList4` iteration

The direct printing of the `ArrayList` has been removed and instead we use the loop in lines 29-31 to call `get` 10 times. That is, once for each value 0 through `theTable.size() - 1`. It is important to note that the first element in the `ArrayList` has the index of 0⁵.

The console output of this program is:

```
~/Chapter07% java -classpath ./usr/lib/jvm/fang.jar PrintAnArrayList4
0 7 14 21 28 35 42 49 56 63
```

⁵This is related to programmers wanting to count from zero. It is not the *reason* but rather a result because `ArrayList` was written by computer scientists.

Notice that the elements are in the same order we inserted them (and that we saw them with `toString` was used to print them). If it were our desire we could call any method in the public interface of the element type.

Updating Elements

Let's modify `IteratedRectangleSprites.java` into an animated program. Let's call it `BoxParade.java` and modify it so that there is an `ArrayList` of `RectangleSprites` containing all of the sprites constructed and added to the game. Then we will use a loop to go across all of the squares, moving them in each call to `advance`.

```

1 import fang.core.Game;
2 import fang.sprites.RectangleSprite;
3
4 import java.util.ArrayList;
5
6 /**
7  * Randomly place some number of rectangle sprites. Then move them
8  * upward at a fixed rate, looping them around off the top of the screen
9  * to the bottom.
10 */
11 public class BoxParade
12     extends Game {
13     /** The collection of RectangleSprites */
14     ArrayList<RectangleSprite> boxes;
15
16     /**
17      * 10 randomly colored and placed rectangles on the screen
18      */
19     @Override
20     public void setup() {
21         // Make sure you initialize the collection!
22         boxes = new ArrayList<RectangleSprite>();
23
24         for (int i = 0; i != 10; ++i) {
25             RectangleSprite curr = new RectangleSprite(0.1, 0.1);
26             curr.setLocation(randomDouble(), randomDouble());
27             curr.setColor(randomColor());
28             addSprite(curr);
29             boxes.add(curr);
30         }
31     }
32
33     /**
34      * Move all the rectangles upward at a fixed speed.
35      */
36     @Override
37     public void advance(double dT) {
38         for (int i = 0; i != boxes.size(); ++i) {
39             boxes.get(i).translateY(-0.5 * dT); // move up
40             if (boxes.get(i).getY() < 0.0) { // loop around at top
41                 boxes.get(i).setY(1.0);
42             }
43         }
44     }

```

45 }

Listing 7.15: BoxParade. java iteration in setup and advance

The important things to notice in the listing: Line 24 calls `new` to create a new collection; this is important because odd errors will occur when calling `.add` in line 31 if it is forgotten. Line 31 is not in `IteratedRectangleSprites`; it adds a reference to the newly created `RectangleSprite` to `boxes`. Lines 41-44 move one box up at a rate of 1 screen every 2 seconds (the `translateY` call is similar to that seen in `NewtonsApple` (see Listing 3.4); the `if` statement is similar to checking if the apple hit the ground but in this case it is if the square hits the top edge of the screen and the reaction is not to randomly place it but rather to “connect” the top and bottom edges of the screen so it looks like the squares wrap-around to the bottom (imagine a cylinder unrolled onto the screen).

An important point here (and a lot of other places in the remainder of the book): whenever you find yourself thinking, “I want to do *blah* to every element in a collection” you should think of a loop. Here, *blah* is “move upward”. In the previous example, `PrintAnArrayList4`, *blah* was “print a space then the element”.

Finding an Element

What else could we do with an `ArrayList`? We could search for a specific element. It might be that we want to know whether a certain value is in the list or we might want to find the minimum or maximum value. Fundamentally this means checking each element in the `ArrayList` against the value being searched for (or the best result found so far). Thus *blah* is “compare to the searched for value” or “compare to the best result so far”.

To put this in action we will modify `BoxParade` so that the element that is closest to the top of the screen will be colored red. When the element closest to the top of the screen changes, we will restore the color of the old highest box and change the appearance of the newly-found highest box.

Assuming we had a way of determining the index of the highest box, we could check if that value has changed inside of `advance`:

```

70     if (currHighest != highest) {
71         boxes.get(highest).setColor(highestColor);
72         highest = currHighest;
73         highestColor = boxes.get(highest).getColor();
74         boxes.get(highest).setColor(getColor("red"));
75     }
76
77     for (int i = 0; i != boxes.size(); ++i) {

```

Listing 7.16: BoxParadeWithRed. java iteration

The code calls (the as yet unwritten) `indexOfHighestBox` which returns the index (integer used with `get`) of the box with the lowest y-coordinate. If that index is different than it was before (as stored in the field `highest`), then we need to replace the color of the *old* highest box, change the `highest` field, save the current color of the highest box (so we can restore it later) and change the color of the highest box to red. Those four steps, in order, are what lines 73-76 accomplish. The last three steps must also be done in `setup` so that the initially highest box is properly colored and has its color saved (this is not listed in the book).

How does `indexOfHighestBox` work? It returns an index or an `int`. This value is initialized to 0 because 0 is a candidate for the highest box on the screen. It is necessary to check every box against *all* of the boxes. Thus we use a `for` loop to go across the `ArrayList`, comparing the location of each box with the best one we have seen so far.

```

51
52     for (int nextIndexToCheck = 1; nextIndexToCheck != boxes.size();
53         ++nextIndexToCheck) {
54         if (boxes.get(nextIndexToCheck).getY() <
55             boxes.get(indexOfHighestSoFar).getY()) {

```

```

56     indexOfHighestSoFar = nextIndexToCheck;
57     }
58     }
59
60     return indexOfHighestSoFar;
61 }
62
63 /**

```

Listing 7.17: BoxParadeWithRed.java iteration

Notice the name of the variables. The value which is returned is the index of the best seen so far. The loop variable is the index of the next one to check (against the best so far). This code returns the highest box seen so far *after* it checks every location in the `ArrayList`. Thus the value returned is the index of the highest box on the screen at the moment.

7.5 ArrayList is an Object

What can we do with a `ArrayList`? Since an `ArrayList` is a standard class type provided by Java, we can look at the JavaDoc documentation for the class and get a list of all of the methods declared within its public interface.

The Public Interface

Much of this section is copied from the JavaDoc page for `ArrayList` in the Java 1.6 release.

Constructor Summary

```
ArrayList<E>()
```

Constructs an empty list of element type `E` with an initial capacity of ten. The initial capacity is the number of times you can call one of the add methods before the class has to move things around in memory to accommodate more elements. An `ArrayList` can contain an arbitrary number of elements.

Method Summary

```
boolean add(E e)
void add(int index, E element)
```

The first version appends the specified element to the end of this list. The second inserts the specified element at the specified position in this list. Either one results in an `ArrayList` with one more element in the list.

```
void clear()
```

Removes all of the elements from this list. No matter how many elements were in the list *before* the call, there will be 0 elements after the call (`size()` returns 0 or `isEmpty()` returns `true`).

```
boolean contains(Object o)
int indexOf(Object o)
```

`contains` returns `true` if this list contains the specified element. It uses `equals()` method to find a match. `indexOf` returns the index of the first (lowest-numbered) occurrence of the specified element in this list, or -1 if this list does not contain the element; `indexOf` also uses `equals` to determine whether or not an element matches the given value. These would not have worked for our `indexOfHighestBox` because they only use `equals` which does not find a minimum or maximum value of a field.

```
E get(int index)
```

Returns the element at the specified position in this list.

```
E set(int index, E element)
```

Replaces the element at the specified position in this list with the specified element. This method returns the value previously at this location.

```
boolean isEmpty()
int size()
```

`isEmpty` returns **true** if this list contains no elements and **false** otherwise. `size` returns the number of elements in this list.

```
E remove(int index)
boolean remove(Object o)
```

The first removes the element at the specified position in this list, returning the value that used to be there (and moving all of the elements with higher indices down one location). The second version removes the first occurrence of the specified element from this list, if it is present. It returns **true** if a matching element was found and removed; **false** if no match could be found. It uses `equals` as defined in the element type to find a match.

These are the basic methods for use with all `ArrayLists`. Summarizing what we know one more time:

- An `ArrayList` is a generic object; the type is named with as `ArrayList<ElementType>` where the `ElementType` is an object type.
- The constructor for an `ArrayList` requires the same generic type specification as a variable declaration.
- An `ArrayList` is an object. This means interaction uses the dot notation we use with other objects.
- `add` adds elements to the list, `get` gets individual elements in the list by index, and `size` returns the current number of elements in the `ArrayList`.
- The indexes of an `ArrayList` start at 0. Thus valid indexes of `ArrayList A` (with any element type) are on the range `[0-A.size())`. As noted previously, 0 is inclusive and the size is not.
- Whenever you think, “I need to *blah* every element in the list,” you should immediately think, “I need to use a loop.”

7.6 Finishing the Flu Simulation

We finally know enough to create our simulation. Given a `Person` class to represent villagers, we can represent the village as an `ArrayList<Person>`. Setup will require adding enough people to the village and selecting some number of them to be the initial victims of the disease.

The simulation will progress until all villagers are either dead or healthy. Each day of the simulation individuals will be told to advance their disease (if any) to the following day; each contagious person will be exposed to some number of their neighbors chosen at random.

The state of the `Person` is similar to the state we saw in `SoloPong` and we will provide useful *predicate* methods. A predicate is a method which returns **true** or **false**; a predicate is also known as a Boolean method. We will also provide transition methods, methods for putting a `Person` into a given state. That way it is easier to translate the state diagram in Figure 7.2 into Java code.

The Person Class Public Interface

Looking at the above description of `Person`, the class’s public interface begins to take shape:

```

public class Person
  extends CompositeSprite {
    Person();

    public boolean isContagious();
    public void makeContagious();
    public boolean isDead();
    public void makeDead();
    public boolean isHealthy();
    public void makeHealthy();
    public boolean isRecovering();
    public void makeRecovering();
    public boolean isSick();
    public void makeSick();

    public void finishRecovery();
    public void expose();
    public void nextDay();

    public void setColor(Color color);
  }

```

The middle 10 methods are related to the five different states that the person can be in. The predicate returns **true** when the person is in the named state and **false** otherwise. This means we need to come up with some way to encode the state of each person (that is an implementation detail; so long as `Person` is treated as an *abstract data type* we don't have to know how that is encoded to use the public interface).

The `setColor` method overrides the method for `CompositeSprite` and colors all the parts of the person with the same color. The state of the person is expressed in the color they display in the village (we can watch who is sick or contagious, etc.).

The three methods above `setColor` are important for the simulation. The first, `finishRecovery` is called to determine whether or not the person got well after their recovery period. That is, in the state diagram, when the person might die or might become healthy, this method makes the appropriate transition. The `expose` method exposes this person to the disease. There is a fixed chance of catching the disease when exposed so a healthy person has a chance of becoming sick when they are exposed. Finally, `nextDay` advances the person one day into the future. If they are sick, the illness runs one more day into its course; if they are dead or healthy, nothing happens. `nextDay` operates on *this* person; an individual `Person` has no idea about finding its neighbors; that is the simulation's job.

Implementation

The implementation of `Person` begins with the definition of a large number of named constants. Recall that named constants are **static final** fields which are assigned once. They can then be used in the code to make the meaning clearer. They also serve as a single place to make changes.

The constants can be divided into three groups. The first group are the states of the `Person`'s health.

```

19 private static final int HEALTHY = 0;
20 private static final int SICK = 1;
21 private static final int CONTAGIOUS = 2;
22 private static final int RECOVERING = 3;
23
24 /** Chance of infection and death */

```

Listing 7.18: `Person.java` health constants

The health will be kept as an `int` field and each of these values will be used to indicate that the person is in a given state. Thus the method `isSick` can just check if `health` (a field of `Person`) is equal to `SICK`.

The second set of constants are the chances of infection and death with the disease along with the number of days the person spends in each stage of the illness. The infection and mortality rates are very high but they match the high end estimates of the flu pandemic of 1918; one reason for clearly marking these values is that the simulation outcome changes a great deal when they are changed.

```

26 private static final double MORTALITY_RATE = 0.20;
27
28 /** Number of days spent in each of the states of health */
29 private static final int DAYS_SICK = 1;
30 private static final int DAYS_CONTAGIOUS = 3;
31 private static final int DAYS_RECOVERING = 3;
32
33 /** Color constants for each of the states of health */

```

Listing 7.19: `Person.java` sickDays constants

Finally, the last group of constants are colors. These colors are the colors the person is displayed with when they are in each of the different states of health. The particular choices here were for shades of green except when the person is dead (grey) or contagious (golden). That makes it easy to follow the spread of the disease.

```

34 private static final Color COLOR_DEAD = Palette.getColor("dark gray");
35 private static final Color COLOR_HEALTHY = Palette.getColor("green");
36 private static final Color COLOR_SICK = Palette.getColor(
37     "green yellow");
38 private static final Color COLOR_CONTAGIOUS = Palette.getColor(
39     "goldenrod");
40 private static final Color COLOR_RECOVERING = Palette.getColor(
41     "yellow green");
42
43 /** Current state of health; drawn from states above */

```

Listing 7.20: `Person.java` color constants

Fields

The fields of a `Person` are the health, the count of days they have been at a given level of sickness, and the body parts displayed for the person.

```

45
46 /** Number of days a sick person has had current state */
47 private int sickDay;
48
49 /** The visible body parts; colored to indicate health state */
50 private final OvalSprite head;
51 private final RectangleSprite body;
52
53 /**

```

Listing 7.21: `Person.java` fields

Initializing a `Person` requires initializing the sprites that make up the body and then initializing health and `sickDays`. Since a person is assumed to be healthy, we reuse `makeHealthy` to avoid having to worry about any extra stuff we need to do to make the new person healthy.

```

59     body.setLocation(0.0, 0.25);
60     head = new OvalSprite(0.5, 0.5);
61     head.setLocation(0.0, -0.25);
62     addSprite(head);
63     addSprite(body);
64     makeHealthy();
65 }
66
67 /**

```

Listing 7.22: Person.java constructor

State Management

The ten middle methods, the state management methods, are all very similar. `isHealthy` and `makeHealthy` will stand for all ten routines.

```

124 }
125
126 /**
127  * Set this person's state to HEALTHY. Update health, sickDay, and
132  health = HEALTHY;
133  setColor(COLOR_HEALTHY);
134  }
135
136 /**

```

Listing 7.23: Person.java *Healthy

As mentioned above, `isHealthy` just checks for equality between `health` and the `HEALTHY` constant. Both the field and the constant are plain-old data so `==` suffices; because `==` returns a Boolean value, the predicate can just return the result of the comparison.

`makeHealthy` is no more involved: the number of days at a given level of illness is set to 0, `health` is set to the right constant, and the color is set to reflect the new health level.

The other eight methods are just the same with the names of the constants suitably changed.

Random Chance

The two methods `finishRecovery` and `expose` use random numbers to decide what happens to this person's health. When they reach the end of their recovery period, `finishRecovery` uses the mortality chance to determine if they die; if they do not die, they recover and are healthy again.

```

178     makeDead();
179     } else {
180     makeHealthy();
181     }
182 }
183
184 /**
185  * This person has been exposed to the disease. Either they get it or
190  (Game.getCurrentGame().randomDouble() < INFECTION_CHANCE) {
191     makeSick();
192     }
193 }

```

```
194
195  /**
```

Listing 7.24: Person. java random chance methods

Similarly, `expose` checks if this person is healthy and if they are it uses the chance of infection to determine if they become sick. Notice that the chance of mortality and infection are numbers between 0 and 1; the lower the number, the less chance that the event occurs. Here a random number between [0.0-1.0) is generated with `randomDouble()` and that value is tested against the chance. If the random number is less than the chance, the chance event happens; otherwise the chance event does not happen.

Day by Day

Finally, the `nextDay` method moves the villager's illness forward one day. That means if they are healthy or they are dead there is nothing to do. If they are sick, `sickDay` is incremented and the number of days they have been at this sickness level is checked against the appropriate `DAYS` constant. If the time has elapsed, then the method to move to the next state is called. Notices that `finishRecovery` is called at the end of the `RECOVERING` state because we don't know whether to make the person healthy or dead.

```
201     ++sickDay;
202
203     if (isSick() && (sickDay >= DAYS_SICK)) {
204         makeContagious();
205     } else if (isContagious() && (sickDay >= DAYS_CONTAGIOUS)) {
206         makeRecovering();
207     } else if (isRecovering() && (sickDay >= DAYS_RECOVERING)) {
208         finishRecovery();
209     }
210 }
211 }
212 }
```

Listing 7.25: Person. java nextDay

Updating the Village

Now that `Person` is implemented, how does a discrete-time simulation work? We need to set up the village: create an `ArrayList` of `Person`, populate it with healthy people, and infect a few of them. Then we need to update the village once per "day": this means using a timer to determine when the number of seconds per day have elapsed. These two top-level methods are the public interface of the simulation game.

Setting up the Village

The listing below shows the fields and the `setup` method of `FluSimulator`. `village` holds all of the villagers. The number of villagers is determined by a constant, `NUMBER_OF_VILLAGERS`; as in `Person`, named constants are used to make the meaning of the value clear. In `setup` the number of villagers determines the number of times `new Person` is called.

```
25  /** The collection of Person representing the village. */
26  private ArrayList<Person> village;
27
28  /** Number of days elapsed since the beginning of the simulation. */
29  private int dayCount;
30
31  /** String sprite assigned to display the day count. */
```

```

32 private StringSprite dayCountDisplay;
33
34 public void setup() {
35     // how big is each villager (so they all fit in one line)
36     double scale = 1.0 / NUMBER_OF_VILLAGERS;
37
38     // set aside space for the collection (no Persons yet)
39     village = new ArrayList<Person>();
40
41     for (int i = 0; i < NUMBER_OF_VILLAGERS; ++i) {
42         // construct one person
43         Person nextVillager = new Person();
44         nextVillager.setScale(scale);
45         nextVillager.setLocation((i * scale) + (scale / 2), 0.5);
46         addSprite(nextVillager);
47
48         // add one person to village
49         village.add(nextVillager);
50     }
51
52     dayCount = 1;
53     dayCountDisplay = new StringSprite();
54     dayCountDisplay.setScale(0.1);
55     dayCountDisplay.leftJustify();
56     dayCountDisplay.setLocation(0.2, 0.1);
57     dayCountDisplay.setText("Day #" + Integer.toString(dayCount));
58     addSprite(dayCountDisplay);
59
60     infectPatientsZero();
61
62     scheduleRelative(this, SECONDS_PER_DAY);
63 }

```

Listing 7.26: FluSimulator.java setup

The number of villagers is also used to determine the scale of each villager's sprite so that they all fit in a single line across the screen. Why is the expression for `scale` `1.0 / NUMBER_OF_VILLAGERS`? Remember that the division operator, `/`, when applied between `int` values, returns the integer quotient. That is, `1 / NUMBER_OF_VILLAGERS` where the number of villagers is greater than 1 will always return the `int` value of 0. By making the first literal in the expression a `double` (by including the decimal point in the literal), the whole expression is treated as a `double` and the number of villagers is changed into a `double` with the same value (so 20 is coerced into the `double` 20.0) and the right value is calculated.

Line 50 uses the loop control variable to calculate the x-coordinate of the location of each villager. The first villager is centered so its left edge touches the left edge of the screen; that mean it's x-coordinate should be half of its width or `scale~/~2`. Each following villager is one "villager width" to the right of the previous one; since `i` starts at 0 and goes up by 1 each time through the loop, `i~*~scale` is the total width of the villagers to the left of villager `i`.

With 20 villagers (number determined by the constant `NUMBER_OF_VILLAGERS`), the game looks like this:

Making an Action

The last line of `setup` is something new. It is a call to a method, `scheduleRelative`. The `scheduleRelative` is a way of scheduling an event to take place some time in the future. FANG schedules the event either at a fixed time after the beginning of the game (using `scheduleAbsolute`) or at some amount of time after the moment the action is scheduled (using `scheduleRelative`). The two parameters to either `schedule` method



Figure 7.5: The FluSimulator

are an Action and a number of seconds. Neither timer runs before the game is started or when the game is paused.

What is an Action? An Action is an *interface* defined in FANG. An **interface** is like a **class** in that it is defined inside of a file with the same name as the interface and all the methods are declared inside of the **interface** body block. An **interface** is different from a **class** in that any class can implement any number of interfaces and an interface provides no “already working” methods or fields. The **interface** just defines a public interface which a class can choose to implement by having methods with the right names and parameter lists. Then it is possible to use a reference to the interface to call methods declared in the interface and implemented by some class. Interfaces will be covered in more detail in the last three chapters of the book.

For the moment all we need to do is note that the Action interface declares a single method, **public void** performAction(). When the scheduled time comes, FANG will call the performAction method automatically.

We could, of course, keep track of elapsed time on our own, counting down some period of time using the dt parameter passed to advance. In fact, that is what we did in the CountdownTimer class in Section 5.2. This time we will let FANG do the timing for us and call our performAction method when the timer expires. performAction will then advance the simulation one day. If the simulation is still running, it will schedule a new event in the future; if the simulation is done running, there is no need to schedule such an event.

```

226 public void alarm() {
227     spreadInfection();
228     advanceSimulatedTime();

```

```

229 // schedule the next update unless the simulation is quiescent
230 if (simulationContinues()) { // schedule a new call in the future
231     scheduleRelative(this, SECONDS_PER_DAY);
232 } else {
233     endgameStatistics();
234 }
235 }

```

Listing 7.27: FluSimulator.java performAction

performAction is a good example of the “pretend it works” approach to top-down design. The problem that it solves is: what to do to advance the simulation’s time clock one day into the future. There are two things to do each day: spread the infection and advance time for the villagers. Then, having advanced time, we can check if the simulation is quiescent; if no health values will change just through the passage of time the simulation is over (all villagers are healthy or dead so no one can become sick, contagious, or recovering ever again).

Leaving spreading the infection for a couple of paragraphs, what do we do to advance time one day for every villager. We want to “advance time one day for every villager in the ArrayList village”. What should you be thinking?

```

186 private void advanceSimulatedTime() {
187     for (int currentVillagerNdx = 0;
188         currentVillagerNdx != village.size(); ++currentVillagerNdx) {
189         Person villager = village.get(currentVillagerNdx);
190
191         villager.nextDay();
192     }
193     // increment the day count and display it
194     ++dayCount;
195     dayCountDisplay.setText("Day #" + Integer.toString(dayCount));
196 }

```

Listing 7.28: FluSimulator.java advanceSimulatedTime

The **for** loop in `advanceSimulatedTime` is formatted across multiple lines. It is customary to break the contents of the parentheses of a **for** loop at the semicolons so that the *<init>*, *<continuation>*, and *<next Step>* are not broken across lines. The body of the loop here just copies a reference to the next villager into the `villager` variable. This works because the entries in `village` are references and the assignment just makes two variables, one inside the `ArrayList` and one declared on line 189 both refer to the same `Person`. Then, using the `villager` reference, `nextDay` is called for the `Person`. As we saw in the last section, `nextDay` advances any existing illness by one day, changing the person’s state (and color) if necessary.

The last two lines in `advanceSimulatedTime` update the time display at the top of the screen (see Figure 7.5 to see the day number message). Advancing simulated time means traversing the entire `ArrayList`, calling a method on every element. Then updating the message on the screen.

Spreading the Illness

To spread the illness, we need to find every contagious villager and have them exposed to their neighbors. The number of neighbors they should be exposed to and how to pick neighbors to expose are left for another pass through the top-down design cycle. That said, the `spreadInfection` method is straightforward:

```

172 private void spreadInfection() {
173     for (int currentVillagerNdx = 0;
174         currentVillagerNdx != village.size(); ++currentVillagerNdx) {
175         Person villager = village.get(currentVillagerNdx);
176         if (villager.isContagious()) {

```

```

177     handleContagiousVillager (currentVillagerNdx);
178     }
179 }
180 }

```

Listing 7.29: FluSimulator.java spreadInfection

We must check each and every villager. If they are contagious, we handle a contagious villager. Handling a contagious villager means picking some number of their neighbors and exposing them:

```

157 private void handleContagiousVillager (int contagiousVillagerNdx) {
158     for (int numberOfExposed = 0; numberOfExposed != EXPOSED_PER_DAY;
159         ++numberOfExposed) {
160         int exposedVillagerNdx = selectNeighbor (contagiousVillagerNdx);
161
162         if (isValidNdx(exposedVillagerNdx)) {
163             village.get(exposedVillagerNdx).expose();
164         }
165     }
166 }

```

Listing 7.30: FluSimulator.java handleContagiousVillager

The loop here, rather than going over all the villagers, runs just EXPOSED_PER_DAY times (convince yourself that this is true by looking at the code). The constant EXPOSED_PER_DAY is set near the top of the simulation to 3; this loop picks three neighbors of each contagious villager and, if the neighbor is actually a valid index into the village, it exposes the neighbor, giving them a chance of becoming sick.

Remember, the model of the illness (the chance of getting sick, what happens when an already sick person is exposed, etc.) is all in the Person and the model of how villagers interact (who is exposed to whom, how to find neighbors, number of villagers, etc.) is in the FluSimulator. This is an example of using different levels of abstraction to overcome complexity. It is also a good example of some bottom-up design in that the Person was written before the game and it has all the right methods already defined.

Selecting a neighbor is done by taking the index of a villager and adding a random number to it. The random number is on the range [-3,3] (where 3 is the value of the NEIGHBOR_DISTANCE constant defined at the top of the game).

```

82 /**
83  * Select a neighbor of the villager at currNdx. Returned value will
84  * be within NEIGHBOR_DISTANCE of currNdx.
85  *
86  * @param currentVillagerNdx the index of the villager for whom we
87  *                            must select a neighbor
88  *
89  * @return an "index" of a neighbor; index in quotes as it might be
90  *         outside the legal range for village
91  */
92 private int selectNeighbor(int currentVillagerNdx) {
93     int neighborNdx;
94     neighborNdx = currentVillagerNdx +
95         randomInt(-NEIGHBOR_DISTANCE, NEIGHBOR_DISTANCE);
96
97     return neighborNdx;
98 }

```

Listing 7.31: FluSimulator.java selectNeighbor

This listing includes the method header comment to show you that it mentions that this method can return “index” values that are out of range. It is important that you document your intent and any limitations of your methods. By including this in the header comment, users of this method know that they are responsible for range checking. You can easily picture what the `isValidNdx` does (called from line 162 inside `handleContagiousVillager`). If a non-valid villager is picked, it is assumed no villager was exposed that time around.

Finishing the Simulation

The simulation is over when health states will not change any more. When is that? If all the villagers were healthy, then none would spontaneously get sick. Similarly, if all the villagers were dead, then none would suddenly get better. The other three health states have out arcs that depend on the number of days in the health state. Thus if any villager is neither dead nor healthy, then we need to keep moving days forward.

```

105 private boolean anySick() {
106     boolean sawSick = false;
107
108     for (int currentVillagerNdx = 0;
109         currentVillagerNdx != village.size(); ++currentVillagerNdx) {
110         Person villager = village.get(currentVillagerNdx);
111         sawSick = sawSick || villager.isSick() ||
112             villager.isContagious() || villager.isRecovering();
113     }
114
115     return sawSick;
116 }
204 private boolean simulationContinues() {
205     return anySick();
206 }

```

Listing 7.32: `FluSimulator.java` `simulationContinues`

`simulationContinues` is just a new name for the `anySick` method. Why? Because the name `simulationContinues` explains what the method is being asked about, the Boolean value it returns. `anySick`, in turn, explains what it does, returning `true` if there are any sick villagers and `false` if there are not. There might well be other uses for the `anySick` method, other than checking if the `simulationContinues`. The separation of the two methods makes reuse of `anySick` simpler.

How can we determine if any villager is sick? We need to check each villager and if they are sick or contagious or recovering, set a flag to `true`. If we start with the flag set to `false` and test every villager, the final value of the flag will be `true` if and only if there is at least one non-dead, non-healthy villager.

This could be done with an `if` statement inside a loop (we have to check every villager). Instead, here, lines 111–112 use a Boolean expression to reset the value of the flag, `sawSick`, each time through the loop. The new value of `sawSick` is `true` if any one of the four Boolean subexpressions is `true`. Once `sawSick` is `true`, it will remain `true` until the end of the method. `sawSick` will *become* `true` on the first villager who is neither dead nor healthy. This is a loop which checks whether *any* element in the `ArrayList` meets some criteria.

When the simulation ends, there is a report on how many villagers died. A new `StringSprite` is created with the information in it and it is added to the game. The interesting part is how we figure out how many villagers are dead. Given `anySick` above, consider how you would approach writing `countOfTheDead`.

```

124 private int countOfTheDead() {
125     int numberOfDead = 0;
126
127     for (int currentVillagerNdx = 0;
128         currentVillagerNdx != village.size(); ++currentVillagerNdx) {
129         Person villager = village.get(currentVillagerNdx);

```

```

130
131     if (villager.isDead()) {
132         ++numberOfDead;
133     }
134 }
135
136 return numberOfDead;
137 }

```

Listing 7.33: FluSimulator.java countOfTheDead

We need to go through every villager and if they are dead, add one to a counter. Initialize the counter to 0 before we start and after we have checked every villager we will have the total number of the dead. Lines 131-133 check if the current villager is dead and increment the counter. The loop control and local variable villager are done just as they are in the other loops in this section.

The only method we have not mentioned yet is `infectPatientsZero`. This is called from `setup` and provides the initial infection. The code is almost identical to `handleContagiousVillager` except that the villagers are picked at random across the whole village and rather than being exposed, each is directly made sick. That way there will always be at least one infected person at the beginning of the simulation. The number of initial patients is determined by the value of the `NUMBER_OF_PATIENTS_ZERO` constant at the top of the simulation.

7.7 Summary

Java Templates

```

<forStatement> ::= for(<init>; <continuation>; <nextStep>) {
                    <body>
                }

```

Chapter Review Exercises

Review Exercise 7.1 How many stars?

Review Exercise 7.2 Name n times?

Review Exercise 7.3 Count examples: by 1's not at 0 start, count by other number

Programming Problems

Programming Problem 7.1 Reimplement TicTacToe with an array; `row(n)`, `column(n)`, `index(r,c)`.

Programming Problem 7.2 Number of people who never became sick

Programming Problem 7.3 Antigens

Programming Problem 7.4 Symmetric social model

Multidimensional Data Structures

In the last chapter we looked at the use of the `ArrayList` type as a collection of objects. The `ArrayList` is limited in that it must hold object types; this is not a big problem because Java provides object “versions” of the built-in plain-old data types: `Integer`, `Double`, and `Boolean` to match the built-ins we have worked with. It also supports automatically using the value of the object versions as the plain-old data types when necessary. Thus if you put a `Boolean` in an `if` statement, Java will automatically convert the `Boolean` into its `boolean` value. It makes the object versions behave “just like” the plain versions.

The big win with `ArrayList` over built-in arrays is that `ArrayLists` are *dynamically* sized; they can grow as necessary to hold whatever number of elements are needed.

This chapter will continue working with `ArrayLists`: this time we will see `ArrayList` with `ArrayList` elements. That is, two-dimensional data structures where a content element is found by getting the *row* in which it resides and then getting the entry at a given *column*.

8.1 Rescue Mission

Consider a job as a lifeguard. A large number of non-swimmers have fallen into the river and are being swept to their doom. Your job is to throw a rescue ring to each, saving the swimmers before they fall off a waterfall. *RescueMission* is just such a game. As shown in Figure 8.1, the swimmers are laid out in a grid. They move back and forth across the screen. Each time a swimmer touches the edge of the screen, the group moves closer to the bottom of the screen. At the bottom of the screen, the rescuer stands, launching a rescue ring into the group in an attempt to save the swimmers. The game is over when a swimmer reaches the bottom of the screen. The game advances to the next level when the last swimmer in the block is rescued: each level is a faster version of the level before.

Above the grid of swimmers is a lane for an occasional “fast swimmer”, something like a swimmer but moving across the screen more quickly than the swimmers and only going across the screen once. Fast swimmers appear at random times, are worth about twice the greatest value of any of the other swimmers, and serve to tempt the player to go for the brass ring.

When the ring is launched, it flies straight up the screen where it was launched. The ring continues until it rescues a swimmer (fast or otherwise) or goes off the top of the screen. Only one ring can be launched at a time. This gives the player a resource management problem to solve: the further away the target is from the rescuer, the longer the player will have to wait to throw the ring again.

This game is interesting for several reasons: given 4 rows of 6 swimmers each, there are almost thirty sprites on the screen at the same time, many more than we have previously handled; all of the sprites (except

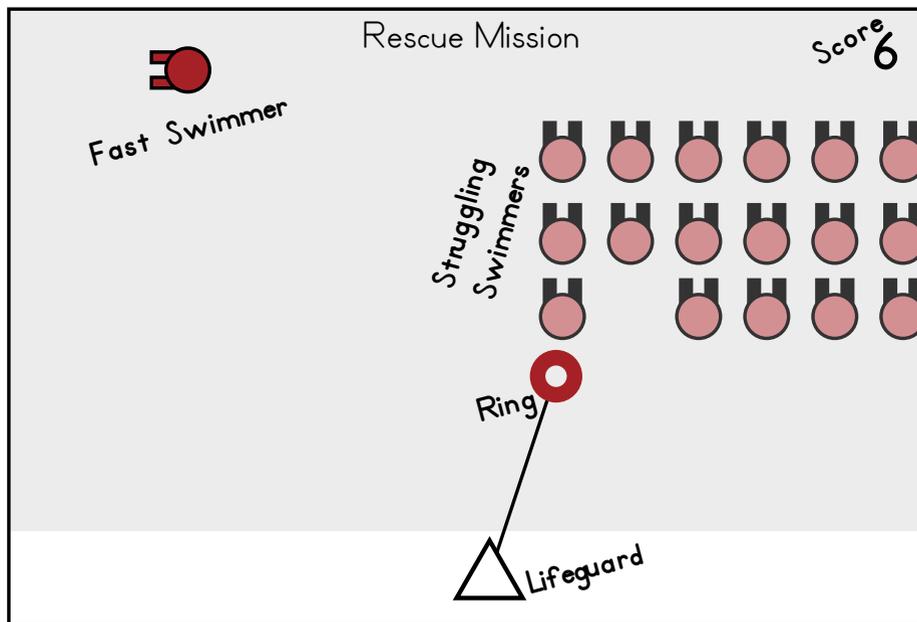


Figure 8.1: RescueMission game design diagram

for the score) move; the swimmers (and the fast swimmer at the top of the screen) are *animated* in that they change appearance while moving across the screen.

Having so many different sprites moving according to so many different rules (swimmers move left-to-right until someone hits the wall; the rescue ring moves up from where it was launched; the rescuer moves left-to-right, controlled by the user's keyboard; the fast swimmer moves from side to side occasionally; the rope is a decoration connecting the rescuer to the rescue ring), it is essential that we divide to conquer. Each sprite class has its own advance method, encoding its own movement rules.

Consider how to call advance for each of thirty sprites. If you look back at Chapter 6 and TicTacToe, you will see that there was a lot of repeating of code to handle just nine sprites on the screen. One argument in favor of using ArrayList collections is that they are composed of multiple elements and we can process each element in them within a loop. Using iteration keeps us from having to repeat ourselves.

We will use iteration but rather than having to somehow translate from a one-dimensional collection (an ArrayList) to a two-dimensional layout (the Swimmer sprites on the screen), we will use a two-dimensional data structure, an ArrayList of ArrayLists of sprites.

Before we get there, though, we will look at the public interfaces of the various sprites in the game and see what we can learn about having lots and lots of moving sprites, each based on the CompositeSprite.

8.2 Inheritance

Looking at the design in Figure 8.1, there appear to be four different classes of sprite: Rescuer, RescueRing, Swimmer, and FastSwimmer. Game implementation begins by examining the public interface of each of these sprites.

Rescuer

Let's start simple: what does the Rescuer need in its interface? Note that we want to be able to have the rescuer move at some speed (a speed we can easily change so that we can tweak our game) according to keyboard input. Does that sound familiar? It should, since Chapter 5's PongPaddle class did almost exactly that. The movement velocity was held in the paddle class which included getter and setter methods for the velocity as

well as a routine to bounce the paddle off the edges of the screen. Using `PongPaddle` as a pattern, `Rescuer`'s public interface looks like this:

```
class Rescuer
  extends CompositeSprite {
  public Rescuer(double deltaX, double deltaY)...
  public void setVelocity(double deltaX, double deltaY)...
  public void setDeltaX(double deltaX)...
  public void setDeltaY(double deltaY)...
  public void advance(double dT)...
  public void bounceOffEdges()...
  public double getDeltaX()...
  public double getDeltaY()...
}
```

Note that there are two ways to set the velocity, either in one dimension at a time or both at the same time. In this program the *y*-coordinate of the velocity will always be zero but as discussed with `PongPaddle`, the more general problem is sometimes easier to solve (and before this section is over, this should be abundantly clear).

The rescuer “bounces” off the edge of the screen in the same way that the `PongPaddle` bounced off the edge: the user cannot move the sprite past the edge of the screen.

Swimmer

Each individual `Swimmer` moves at a given speed first one way and then another way across the screen. Again, reaching back to `SoloPong`, this is a lot like the ball, moving in a straight line until it bounces off of the edge of the screen. The public interface of `Swimmer` looks like this:

```
class Swimmer
  extends CompositeSprite {
  public static void bumpTheWall()...
  public static void clearBumpTheWall()...
  public static void hasBumpedTheWall()...

  public Swimmer(int score, double deltaX, double deltaY)...
  public void setVelocity(double deltaX, double deltaY)...
  public void setDeltaX(double deltaX)...
  public void setDeltaY(double deltaY)...
  public void advance(double dT)...
  public void bounceOffEdges()...
  public double getDeltaX()...
  public double getDeltaY()...

  public void setScore(int score)...
  public int getScore()...
}
```

The *individual* swimmer interface is similar to that found in `Rescuer`; by individual we mean non-**static**. The **static** methods are there so that *all* of the `Swimmer` objects reverse direction when *any* swimmer touches the edge of the screen.

The score value of a `Swimmer` is how many points the swimmer is worth when rescued. It is variable so that each row of swimmers can have a higher value than the one below it. The exact score will be determined in the game when the swimmers are constructed.

RescueRing

The RescueRing is either waiting to be thrown or it has been thrown. If it is waiting to be thrown, it should simply move along with the Rescuer; if it has been thrown, then it moves according to its current velocity until it rescues a given Swimmer or it hits the edge of the screen. Thus the public interface is:

```
class RescueRing
  extends CompositeSprite {
  public RescueRing(double deltaX, double deltaY)...
  public void setVelocity(double deltaX, double deltaY)...
  public void setDeltaX(double deltaX)...
  public void setDeltaY(double deltaY)...
  public void advance(double dT)...
  public void bounceOffEdges()...
  public double getDeltaX()...
  public double getDeltaY()...
}
```

FastSwimmer

The FastSwimmer is a Swimmer turned on its side. The big difference is that the FastSwimmer has two states: swimming and waiting. When it is waiting, each frame it determines whether or not it is time to launch and when it is swimming it just moves according to its current velocity. That means that the public interface for FastSwimmer is the same as that of Swimmer. In fact, a FastSwimmer is a Swimmer so the public interface is:

```
class FastSwimmer
  extends Swimmer {
}
```

FastSwimmer will override all of the non-static member functions of Swimmer because they perform slightly differently (in particular, FastSwimmer has no impact on bouncing the rest of the swimmers off walls and fast swimmers don't bounce off the edge, really; fast swimmers go to the waiting state when they hit an edge).

The following screenshot shows the grid of swimmers as well as the fast swimmer zipping across the top of the screen.

Is-a

The emphasis in the sentence above, that FastSwimmer is a Swimmer, indicates a special relationship between the two classes: every FastSwimmer is first a Swimmer and then something more. This relationship is so common in object oriented programming that computer scientists have turned it into a single word: *is-a*. When one class extends another, any child class object is-a parent class object.

What this means is that anywhere where a parent class object is expected, you can replace it with a child class object. Consider a landscaping plan. If there is to be a tree at the north-east corner of the yard, it is possible to plant an apple tree there. This is because an apple tree is-a tree. There are things you can do with an apple tree which you cannot do with any arbitrary tree: sit under it and discover gravity, pick an apple; there is nothing you can do with an arbitrary tree that you cannot do with an *apple* tree: plant it, prune it, water it, etc.

Extracting Common Functionality

Looking back over the three public interfaces for Rescuer, RescueRing, and Swimmer, the three of them have much in common. All three classes have velocity; that implies each has two fields for holding the x-component and the y-component of the velocity in addition to the five routines for getting and setting the velocity. More



Figure 8.2: A Screenshot of RescueMission with a FastSwimmer

than that, all of the classes do the “same thing” in advance, at least most of the time: translate the current position of the sprite by the velocity (in screens/second) times the frame delay time (in seconds). The DRY Principle dictates that we should avoid repeating ourselves. Thus we want to have some way to factor the two fields and their associated methods along with `advance` and `bounceOffEdges` out of all three classes and define them once rather than three different times.

How do classes share implementation? That is, how can two classes use the same method? We have been using this capability since the beginning of the book: any class which extends another inherits the parent class’s public interface. So, to have a single copy of the fields and common definition of most of the methods, all three classes must extend a class with those methods in the public interface. If we call the missing link in this hierarchy `SpriteWithVelocity`, we split the three classes into four with the following public interfaces:

```
class SpriteWithVelocity
  extends CompositeSprite {
  public SpriteWithVelocity(double deltaX, double deltaY)...
  public void setVelocity(double deltaX, double deltaY)...
  public void setDeltaX(double deltaX)...
  public void setDeltaY(double deltaY)...
  public void advance(double dt)...
  public void bounceOffEdges(...
  public double getDeltaX(...
  public double getDeltaY(...
```

```

}

class Rescuer
  extends SpriteWithVelocity {
  public Rescuer(double deltaX, double deltaY)...
}

class RescueRing
  extends SpriteWithVelocity {
  public RescueRing(double deltaX, double deltaY)...
}

class Swimmer
  extends SpriteWithVelocity {
  public static void bumpTheWall()...
  public static void clearBumpTheWall()...
  public static void hasBumpedTheWall()...

  public Swimmer(int score, double deltaX, double deltaY)...
  public void setScore(int score)...
  public int getScore()...
}

```

The number of classes is one higher than it was before but each of the subclasses of `SpriteWithVelocity` is simpler. All the classes will override `advance` and `bounceOffEdges` but each class is simpler because of the *abstraction* of the velocity out of several classes. With comments, `SpriteWithVelocity.java` is just under 200 lines of code; this code would have to appear in each of the three classes which now inherit it.

This is primarily an example of sharing implementation and features through inheritance. While each `Swimmer` is-a `SpriteWithVelocity` (and the same for `Rescuer` and `RescueRing`), we are not using that fact. FANG does use an is-a relationship in the heart of the video game loop.

Displaying Sprites

As a reminder, the main video game loop in FANG is

```

setup
while (not game over)
  displayGameState
  getUserInput
  advance

```

The part we are interested is `displayGameState`. How does FANG display all the different sprites on the screen? Inside of FANG is a class called `AnimationCanvas` which has a field called `sprites` of type `ArrayList<Sprite>`¹. Each time through the video game loop, `AnimationCanvas`'s `paintSprites` method is called. It is called with an object of type `Graphics2D`; a `Graphics2D` represents the current window on the video screen where the program is running. The `paintSprites` method looks something like this:

```

class AnimationCanvas ... {
  ...
  ArrayList<Sprite> sprites;
  ...
}

```

¹The *actual* type of `sprites` is more complicated than `ArrayList`; it is much more similar to an `ArrayList<ArrayList<Sprite>>` where each `ArrayList<Sprite>` is the collection of sprites entered with the same z-ordering or layering number. Thus what we will present as a single loop is actually a nested pair of loops, one selecting the layer and the inner one selecting each `Sprite` on the layer (and even this is a simplification).

```
private void paintSprites(Graphics2D brush) {
    for (int i = 0; i != sprites.size(); ++i) {
        sprites.get(i).paintInternal(brush);
    }
}
...
}
```

What is special about this? It is an example of code using the is-a relationship. The elements of `sprites` are of type `Sprite` as far as this code knows. You know that what you called `addSprite` with was a `RectangleSprite` or a `Swimmer` or a `StringSprite`. The `Sprite` class provides a method called `paintInternal`; each specialized type of `Sprite` provides a version of `paintInternal` which knows how to put a rectangle or an oval or an image or even an entire composite collection of other sprites onto the screen. That way the specialized actions of each of the special types of `Sprite` take place even while all `AnimationCanvas` needs to know is that all `Sprite` objects know how to `paintInternal`.

In the landscaping example, imagine each tree knows how to `makeBlooms`. To have the whole garden bloom, it is enough to cycle through all of the trees and call `makeBlooms` for each tree. The apple trees will make apple blossoms, cherry trees will make cherry blossoms, and pine trees might not actually have flowers in the more traditional sense so they might do nothing.

The is-a relationship is the power we leverage in using abstraction to tame complexity. By separating the details of how each `paintInternal` method works from the details of figuring out when `paintInternal` is called for each `Sprite`, we limit the details we must remember at any given moment.

The next section looks at how we can have collections which hold collections and the section after that discusses animated sprites. After that we will come back to the `RescueMission` game and look at how to use `SpriteWithVelocity` to share several useful methods among multiple classes.

8.3 Multidimensional Collections

Before we work on declaring `ArrayLists` of `ArrayLists` (that is where this is going), let's take a quick look at two *nested loops*. Iteration such as “Remove the smallest numbered card from this deck of numbered cards, placing it on the table in front of you,” can, itself, be iterated as in “Remove one smallest card (as described previously) once for each card in the deck.” The algorithm described is something like (this is *not* actual Java):

```
for(i = 0; i != #of cards; ++i) {
    for (j = 0; j != #of cards still in deck; ++j) {
        if this one is smallest see so far, grab it
    }
    Lay the card you grabbed on the table
}
```

What is the result of following the above instructions? The cards are placed on the table in *ascending order*. That is, the deck of numbered cards is sorted when this algorithm finishes. We will come back to this sorting idea later. For the moment just note that the idea of iterating iteration is not as foreign to you as it might seem.

Nested Loops

Remember that the body of a `for` loop can be any Java code that is valid in a method body². This means that it is legal to have a loop inside of another loop.

Consider the following code. How many asterisks are printed?

²Java does not support nested method or type declarations. You cannot define a method *inside* the body of another method nor can you define a `class` *inside* the body of a method. The same is true of code inside a `for` loop; the `for` loop must, itself, be inside of some method.

```

for (int i = 0; i != 5; ++i) {
    for (int j = 0; j != 3; ++j) {
        System.out.println("*");
    }
}

```

Looking at the second `for` line and the `<init>`, `<continuation>`, and `<nextStep>` parts, we can see that `j` takes on all the values from `[0-3)` or 0, 1, and 2. When the loop control variable begins at 0, the next step increments the loop control variable and the comparison uses `!=` (or `<`) some number, n , the body of the loop is run n times. This is the *idiomatic* way to write a count-controlled loop, a loop which runs a certain number of times. This is the way a Java programmer writes such a loop; it makes it easy to read and to reason about the loop.

How many times is the body of the outer loop executed? Reading the `for` line, it, too, is an idiomatic count-controlled loop. That means it runs 5 times.

The inner loop body draws a asterisk each time it is executed. Thus each time the inner loop is executed (in total) it draws 3 asterisks. Since the inner loop is the body of the outer loop, it is executed 5 times. Thus 5 times (once for each loop iteration) there are 3 asterisks drawn; this means there are 15 asterisks drawn, one per line (each is drawn using `println` which starts a new line).

2-Loops, 2-Dimensions

The above code prints out a single, vertical line of asterisks; how would you change it to print out all of its asterisks on a single line and only after printing all of them start a new line? That would mean changing the `println` to `print` (so it does not start a new line) and then adding `System.out.println()` after the end of the loops. All 15 asterisks in a single line.

```

for (int i = 0; i != 5; ++i) {
    for (int j = 0; j != 3; ++j) {
        System.out.print("*");
    }
}
System.out.println();

```

What code would print out 5 lines of 3 asterisks each? In this case we want to use `print` in the inner loop (to keep the asterisks on a single line) and then, whenever the outer loop is about to end, start a new line with the `System.out.println()`.

```

for (int i = 0; i != 5; ++i) {
    for (int j = 0; j != 3; ++j) {
        System.out.print("*");
    }
    System.out.println();
}

```

If you use the fact that we know, because the loop is written idiomatically, the number of times the outer loop executes is 5. That means, because `println` is inside the loop (and not inside of any other loop), this code prints 5 lines. Since each line also has whatever is printed by the inner loop, each line has 3 asterisks.

Multiplication Table

How would you write a game that displayed a multiplication table on the screen? This is different than printing asterisks because we want to position things on the screen. It is similar, though, because we need to process something with two dimensions. We will start displaying a simple multiplication table. Then we will look at how to put references to the displayed elements into a 2-dimensional data structure and then we will see how to go through all the elements in the structure so we can highlight some of the elements by changing their colors.

```

8  /** The number of columns of numbers in the table */
9  public static final int COLUMNS = 10;
10
11 /** The number of rows of numbers in the table */
12 public static final int ROWS = 6;
13
14 /** StringSprite + space between them */
15 public static final double SPACING = 1.0 /
16     (1 + Math.max(ROWS, COLUMNS));
17
18 /** Scale of StringSprites (for leaving space) */
19 public static final double ACTUAL_SCALE = SPACING - 0.02;
20
21 /** Color for entries labeling the table */
22 public static final Color LABEL_COLOR = getColor("misty rose");
23
24 /** Offset from edge to labels (on top or left) */
25 public static final double LABEL_OFFSET = SPACING / 2.0;
26
27 /** Color for entries in the table */
28 public static final Color TABLE_COLOR = getColor("yellow green");
29
30 /** Offset to first row or column (top/left) */
31 public static final double TABLE_OFFSET = LABEL_OFFSET + SPACING;

```

Listing 8.1: MultiplicationTable.java constants

MultiplicationTable begins with a collection of constant values. The number of rows and columns, the size of each StringSprite and the size in which each sprite is positioned (the scale of the sprite is smaller so there is space between entries), the offset of the first entries in the tables and the first entries of the label rows and columns are calculated based on the spacing of the entries. Finally, the colors of the table entries and the labels are specified. Notice that all of the constants are **static** meaning there is only one copy, no matter how many MultiplicationTable objects are instantiated, and **final** meaning that they can only be assigned to once. By assigning to the **static final** values at the same time they are declared makes it easier for others to come along later and change the constant values; if we make consistent use of ROWS, for instance, changing it from 6 to 7 should change everything so that 7 rows are drawn in the right scale.

```

37 public void setup() {
38     labelRows();
39     labelColumns();
40     fillProductTable();
41 }

```

Listing 8.2: MultiplicationTable.java setup

The listing above is another example of the power of top-down design. setup just calls three routines, labelRows and labelColumns to place the row and column labels on the screen and fillProductTable which places all of the products.

```

74 private void labelRows() {
75     double yOffset = TABLE_OFFSET;
76     double xOffset = LABEL_OFFSET;
77     for (int row = 0; row != ROWS; ++row) {
78         makeOneEntry(xOffset, yOffset, row, LABEL_COLOR);
79         yOffset += SPACING;

```

```

80     }
81 }

```

Listing 8.3: MultiplicationTable.java labelRows

labelRows serves to explain both label methods. There is a count-controlled **for** loop which runs ROWS times for numbers in the range [0-ROWS). It creates a `StringSprite` each time through the loop and it changes the `yOffset` by the spacing value each time through the loop. Thus each `StringSprite` is placed directly below the one before it on the screen (the `xOffset` does not change). The text inside of each `StringSprite` is the value of the loop control variable.

How do we know the value of the text? Look at the body of `makeOneEntry`:

```

96 private StringSprite makeOneEntry(double x, double y, int value,
97     Color color) {
98     StringSprite tableEntry = new StringSprite();
99     tableEntry.setScale(ACTUAL_SCALE);
100    tableEntry.setLocation(x, y);
101    tableEntry.setColor(color);
102    tableEntry.rightJustify();
103    tableEntry.setText(Integer.toString(value));
104    addSprite(tableEntry);
105    return tableEntry;
106 }

```

Listing 8.4: MultiplicationTable.java makeOneEntry

Setting the color, scale, and location are typical for creating any sprite. The `rightJustify` method sets the position of a `StringSprite` to the right edge of the string. This means the unit digits will align vertically. What is the text set to in line 103? The value parameter is of type `int`; this is *not* an object type. Thus there is no way to call `value.toString` (you cannot apply the dot operator to non-object types). The object type, `Integer`, provides a **static** method, `toString` which takes a single `int` as its parameter and it converts the value to a string.

TK

Declaring a 2-Dimensional ArrayList

Now that we have some feeling for how to use nested loops to work with two dimensions of data, we will now look at how to declare a two-dimensional data *structure*, an `ArrayList` of `ArrayList`s. This section will focus on creating a two-dimensional list of `Swimmers`; for the moment we will only be using the objects' `Sprite` interface functions though we will declare the array list to hold `Swimmer` objects.

```

20 public static final int COLUMNS = 6;
21
22 /** add to level for fast swimmer's score */
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65 /** The water background */

```

Listing 8.5: RescueMission: Declaring swimmers

Listing 8.5 shows the declaration of `swimmers` inside the `RescueMission` class (`RescueMission` extends `Game`). The two **static final ints** are the number of rows and columns of sprites we will be declaring. Having named constants and using them consistently means changing these numbers in lines 21 or 22 changes the number of swimmers throughout the program.

Just as before, the declaration of an `ArrayList` type consists of the word `ArrayList` followed by the element type in angle brackets. Unlike in the last chapter, here the elements of `swimmers` are not sprites but rather are *lists of sprites*. That is, after it is properly initialized, `swimmers.get(0)` will return the first `ArrayList<Swimmer>` in the field.

An `ArrayList` can have any object type as its element type and `ArrayList<...>` is an object type, so it is legal to have elements which are, themselves, `ArrayList`s. Inside the angle brackets must be a complete object type name; `ArrayList` is a *generic type* so to be a complete type we must provide the element type of the collection. We want the inner collections to contain `Swimmer` objects. This is just what the declaration on line 65 does.³

Having declared a field which holds lists of lists, how do we initialize it? Obviously at some point we need to call `new` and set the value of the field to the result. What constructor will we call? Then the question is what do we pass to add to add new elements to the list of lists.

```

347     final double lowestRowY = swimmerOffset + (ROWS * objectScale);
348     swimmers = new ArrayList<ArrayList<Swimmer>>();
349     for (int row = 0; row != ROWS; ++row) {
350         ArrayList<Swimmer> currentRow = new ArrayList<Swimmer>();
351         double rowY = lowestRowY - (row * objectScale);
352         int rowScore = row + 1; // higher row = higher score
353         for (int column = 0; column != COLUMNS; ++column) {
354             Swimmer current = new Swimmer(rowScore, swimmerDX, swimmerDY);
355             current.setScale(objectScale);
356             current.setLocation(swimmerOffset + (column * objectScale),
357                               rowY);
358             addSprite(current);
359             currentRow.add(current);
360         }
361         swimmers.add(currentRow);
362     }
363 }
364
365 /**

```

Listing 8.6: `RescueMission`: `setupSwimmers`

The `setupSwimmers` method from `RescueMission` sets up the list of lists and the elements in it. Line 350 calls `new` to construct `swimmers`; the constructor is of type `ArrayList<ArrayList<Swimmer>>`. Ignoring lines 353-362 for a minute, we can see that line 352 constructs a new `ArrayList<Swimmer>` and line 363 adds that new list to `swimmers`.

How many times are lines 352 and 363 executed? The loop control variable, `row` is initialized to 0, is incremented each time through the loop, and the loop is exited when `row` is `ROWS`. That means the loop runs across the integer range `[0-ROWS)`. Thus the number of rows added is `ROWS` (set to 4 as we saw above in line 21).

What happens when we stop ignoring lines 353-362? Line 353 calculates the y-component of all sprites on the current row. The first row is the lowest row on the screen, the second the next one up, and so on. Thus line 349 calculated the y-component of the lowest row (the *highest* y value because of the inverted y-axis). Each row we subtract row number times height of a row from that value. Similarly, the score for rescuing swimmers in the lowest row is 1, the second row up is 2, and so on for each row. Thus the score for all swimmers in a row is calculated in line 354.

The loop in lines 355-362 is executed with `column` across the integer range `[0, COLUMNS)`, or `COLUMN` times (set to 6 in line 22 above). Inside the loop we create one `Swimmer` sprite. That sprite is scaled, located, added to the game (so it is in the list used in `AnimationCanvas`) and added to the current row's `ArrayList`. Thus 6 `Swimmers` are added to each row and 4 rows are added to `swimmers`: there are 24 `Swimmer` sprites added to the game.

We could have added any number of sprites to the game with `addSprite`. This code is powerful because we retain the ability to access each individual `Swimmer` sprite because we have the collection of collections of `Swimmers`.

³In this book we will not include spaces within the angle brackets. Thus the type of `swimmers` is `ArrayList<ArrayList<Swimmer>>`. It should be noted that spaces are permitted within and around the angle brackets and some sources recommend using `ArrayList< ArrayList< Swimmer >>` as the spacing for the type name. This is mentioned here so readers are aware of it when they see other code samples.

Traversing a 2-Dimensional ArrayList

How can we use `get` to get each `Swimmer`? In the last chapter we saw that to do *anything* with every element in an `ArrayList` means to use a count-controlled loop, iterating across all the elements and using the `get` method to retrieve each entry.

That suggests the following model of code for calling `advance` on each and every `Swimmer` in `swimmers`:

```
for (int row = 0; row != ROWS; ++row) {
    ArrayList<Swimmer> currentRow = swimmers.get(row);
    for (int column = 0; column != COLUMNS; ++column) {
        Swimmer curr = currentRow.get(column);
        // do something with curr here
    }
}
```

This is *exactly* what is done in the `moveEverything` method of `RescueMission` except that instead of having a variable holding a reference to the current row we just use `swimmers.get(row)` whenever we need the current row.

```
227     rescuer.advance(dT);
228     ring.advance(dT);
229     for (int row = 0; row != ROWS; ++row) {
230         for (int column = 0; column != COLUMNS; ++column) {
231             Swimmer curr = swimmers.get(row).get(column);
232             if (curr != null) {
233                 curr.advance(dT);
234             }
235         }
236     }
237 }
238
239 /**
```

Listing 8.7: `RescueMission: moveEverything`

In `moveEverything`, the fast swimmer, `rescuer`, and rescue ring are moved first (lines 228-230); we will return to these objects a little later but just consider that `advance` just moves each one according to its movement rules outlined above. The two nested loops look just like the ones shown in the previous snippet except for line 219 which uses two chained calls to `get`. The key thing to note is that `get` returns an element of the `ArrayList` no matter what type the elements have. Thus line 233 is parsed as follows:

```
Swimmer curr = swimmers.get(row).get(column);
               = (swimmers.get(row)).get(column);
```

Thus the result of `swimmers.get(row)` is an `ArrayList<Swimmer>` (just like in the earlier snippet) and applying `get` to that array list yields a `Swimmer` just like the previous snippet.

What about the `if` statement inside the loop? In the previous section we saw that we added the results of calling `new Swimmer` twenty four times; how can any element be `null`?

At the beginning of the level no entry *can* be `null`. The question is, what happens when a swimmer is rescued? The game is played by launching the rescue ring and if the rescue ring intersects a swimmer, that swimmer is rescued and the score for that swimmer is added to the player's score. How is the rescue of the `Swimmer` in the 2D `ArrayList` indicated?

There are two possible answers: We could use the `hide` method defined in `Sprite` to turn the swimmer invisible or we could actually remove the swimmer from the array list by putting a `null` reference in its place. The `if` statement should make it clear which design decision was made in this program. We will briefly examine the path not taken and then look at the implications of `null` references in our grid.

Using hide. When a Sprite is hidden, it remains on the screen and it will return `true` if there is an intersection test with any other sprite where the visual representation of the hidden sprite *would have* intersected with the other, had it been visible. That is, hidden sprites still intersect with other sprites. They also have locations and bounds so it is possible for them to intersect with the edges of the screen.

A hidden sprite is still “there” and an additional check is required with each intersection test to have our game pretend that they are not. The advantage to using `null` is that the Java Virtual Machine will yell if we dereference (use the dot operator on) a `null` reference while it will not yell if we forget to check whether a given sprite is hidden. While errors, especially run-time errors, are annoying, in this case they make finding missing reference checks easier to find. In the other case we might just see odd bouncing behavior but be hard pressed to narrow down where to look for it.

There is a set method for `ArrayList` objects which takes an index (just like `get`) and a new value to put in the `ArrayList` at that location. Thus rescuing the Swimmer at the position (`row`, `column`) in the grid is done by `rescueSwimmer`:

```

297     setScore(getScore() + curr.getScore());
298     removeSprite(curr);
299     swimmers.get(row).set(column, null);
300     ring.startReady();
301 }
302
303 /**

```

Listing 8.8: RescueMission: rescueSwimmer

The FANG `removeSprite` method is analogous to `addSprite` but it removes the reference sprite from the list used inside of `AnimationCanvas`. Then we update the location where the sprite was to contain `null`. Notice that we do not remove rows, just individual Swimmers.

Looking at the description of the game, a level ends when the player rescues all of the swimmers in the grid and the game ends when a swimmer reaches the bottom of the water. Detecting each of these conditions requires *traversing*⁴ across the `swimmers` collection. Each does something a little bit different with each element.

Counting the Remaining Swimmers

A level is finished when the last swimmer, from `swimmers`, is rescued. This means the `FastSwimmer` does not count for ending the level. How can we determine how many swimmers remain? We can count how many non-`null` references remain in `swimmers`. Again, doing *anything* with every element in an `ArrayList` means we should be thinking about a count-controlled loop. For a two-dimensional collection, we should be thinking about two *nested*, count-controlled loops.

```

248     for (int row = 0; row != ROWS; ++row) {
249         for (int column = 0; column != COLUMNS; ++column) {
250             Swimmer curr = swimmers.get(row).get(column);
251             if (curr != null) {
252                 ++numberOfSwimmers;
253             }
254         }
255     }
256     return numberOfSwimmers;
257 }
258
259 /**

```

Listing 8.9: RescueMission: remainingSwimmerCount

⁴“to travel or pass across, over, or through” [AHD00]

The `remainingSwimmerCount` method starts by initializing the count to 0. Then, in a pair of nested loops, the outer looping over rows and the inner looping over the columns `curr` is set to refer to each `Swimmer` in the structure, one after the other. The `if` statement (which should look very familiar) tests if `curr` is not `null`. As long as the reference isn't `null` the swimmer counts. Thus the body of the `if` statement increments the return value. Then, when the loops finish, the count is returned.

Examining this method now is an example of bottom-up design: we know we will need this ability to test for the end of a level. Now we have a command that gives us the swimmer count. We will look at how to launch a new level later in this chapter.

Finding the Lowest Swimmer

The *game* is over when the lowest row of swimmers remaining on the screen reach the bottom of the water. That is, when the lowest y-coordinate of one of the lowest `Swimmer` sprites is greater than or equal to the bottom of the water. If we could get a reference to one of the lowest sprites in the `swimmers` collection, the Boolean expression testing for the game being over is straight forward:

```
((getLowestSwimmer() != null) &&
 (getLowestSwimmer().getMaxY() >= water.getMaxY()))
```

Now, how does `getLowestSwimmer` work? It should return a reference to any `Swimmer` in the lowest row on the screen. Looking back at `setupSwimmers`, we find that row 0 is the lowest row. Unfortunately, it is possible the player rescued *all* the swimmers in the bottom row (they are all set to `null` as discussed above; we have not seen the rescue code yet but we know it should `null` out the rescued swimmer).

What we need to do is go through the list of lists and check for the first non-`null` value. Again, we should be thinking of a nested pair of count-controlled loops.

```
214 }
215
216 /**
```

Listing 8.10: `RescueMission`: `getLowestSwimmer`

This pair of loops should look familiar: the loop control variable is initialized to 0 and incremented until it reaches the end of that dimension's size. The Boolean expression in the middle of the `for` loop is extended, though: these loops also end as soon as a non-`null` swimmer is found. So, assuming all of the `swimmers` remain, when both `row` and `column` are 0, the `if` statement on line 210 evaluates to `true` so `lowest` is set to `curr` (or `swimmers.get(0).get(0)`). This is not `null` so the `(lowest == null)` expression is `false` and each loop ends early (remember: `false` and anything is `false`).

Bottom-up Design

Using `remainingSwimmerCount` and `getLowestSwimmer` as building blocks lets us write two Boolean methods, `checkIsGameOver` and `checkIsLevelOver`. Then in advance we can call these methods in `if` statements. Notice that these two methods are quite simple; the reason for writing them as methods is to make the selection statements “self-documenting”⁵.

```
183     (getLowestSwimmer().getMaxY() >= water.getMaxY()));
184 }
185
186 /**
191 }
192
```

⁵The scare quotes are here because some programmers argue that properly written code is always self-documenting while others claim that such a term is an oxymoron. As discussed in earlier chapters, the current authors believe in a balance between comments which document the programmer's *intent* and the use of good names and an appropriate level of abstraction in support of readable code.

193 /**

Listing 8.11: RescueMission: checkIs...

Checking for the game being over is easy: `checkIsGameOver` evaluates the Boolean expression given earlier, returning `true` if there is a surviving swimmer *and* that swimmer has reached the bottom of the water.

The Boolean expression should have you scratching your head. We know that `false` and anything (`true` or `false`) is `false` but if `(getLowestSwimmer() != null)` is `false` then `getLowestSwimmer()` must return `null`. That means the expression after the `&&` will throw a *null-pointer exception*. That is, `getLowestSwimmer().getMaxY()` cannot be evaluated without causing the program to crash.

Java uses the fact that `false` and anything is `false` to avoid this problem. Using something called *short-circuit Boolean evaluation*, Java evaluates a complex Boolean expression only until it knows the result. Expressions at the same level are evaluated from left to right. If the expressions are joined by `&&` (such an overall expression is called a *conjunction*), as soon as Java evaluates one expression as `false` it can stop evaluating the expression because the result is known.

Similarly, with Boolean expressions joined by the `||` operator (called a *disjunction*), as soon as Java evaluates one expression to `true` it can stop evaluating as the overall expression's value must also be `true`. Many modern languages use short-circuit Boolean evaluation to permit tests to make sure expressions are safe to evaluate (what we do in line 142) and to speed up programs (if the answer is known, why do any more processing to evaluate the expression?).

`checkIsLevelOver` just counts the number of swimmers remaining and compares that number to 0; if there are 0 swimmers, all have been rescued and the current level is over. We will get back to having multiple levels after a slight diversion into animation and how we can animate our `Swimmer` class.

8.4 Animation

Animation, “the act, process, or result of imparting life, interest, spirit, motion, or activity” [AHD00], is nothing new. From the falling apple in `NewtonsApple` to the color-changing villagers in the `FluSimulator`, all of the games we have written since Chapter 2 have added interest through motion and activity. This section reviews what we have done before and talks about how to animate `Swimmer` sprites in `RescueMission`.

Smooth Move: Animating Sprites

At its core animation, be it computer animation or traditional painted celluloid animation, is really a loop. If you look at film and a film projector you will find that the film is a long stream of pictures, each printed on the celluloid at a fixed distance from the previous and following pictures. The projector aligns the film so that one of the pictures is projected through the optics. When the projector is running, a given picture is held for a short, fixed amount of time. Then, for a much shorter amount of time, a shutter blocks the projection of the light through the film and, at the same time, the film is moved to the next frame.

That sequence: show frame-update frame sounds familiar. It is, in fact, the first and last steps of the video game loop:

- **Display the current game film state**
- `Get user input`
- **Update game film state according to current state and user input**

This means that we can, already, see how we should include animation inside our objects: advance in our various sprite classes will enforce movement rules according to the game and they will also change the appearance of the sprite at some rate.

Making use of `advance` on the Objects

Looking at `moveEverything` in Listing 8.7, we see that `advance` is called for every `Swimmer` in the grid to make it move. If each swimmer should wave their arms back and forth in an effort to get the player's attention, that is where we should put the code to animate the sprite.

Let's look at `Swimmer.advance`:

```

13 private static boolean bumpedTheWall = false;
98 public void advance(double dT) {
99     super.advance(dT);
100     decrementTimer(dT);
101     if (runIntoLeftWall() || runIntoRightWall()) {
102         bumpTheWall();
103     }
104 }

```

Listing 8.12: `Swimmer.advance`

Lines 13-14 are included in the listing to remind us that a `Swimmer` is-a `SpriteWithVelocity`. Thus in `advance` on line 99 the call to `super.advance(dT)` is a call to the `advance` method provided by that class. Whenever possible we will use the commonly defined implementations of methods from `SpriteWithVelocity` to justify our extracting the code. `SpriteWithVelocity.advance(double)` has just one line in the body:

```

9  /** The sprite's velocity */
10 private double deltaX;
42 }
43
44 /**

```

Listing 8.13: `SpriteWithVelocity.advance`

Generally, if a sprite has velocity, it typically is advanced by moving it according to how long it has been from the last frame and how fast it should be moving.

Lines 101-103 handle bumping into the wall. The `bumpTheWall` method is **static**. What does **static** mean again? There is no **this**. The method and the field it manipulates are shared by *all* `Swimmers`. If *any* `Swimmer` touches the wall, then *all* `Swimmers` will see the change in the shared field. More on this in a minute.

Line 100 handles animation. That is, each `Swimmer` has a timer. When the timer runs out, then the *state* of the `Swimmer` is updated. This should sound a lot like how `FluSimulator` ran except that each sprite has its own timer rather than using a `FANG` timer class.

```

27
28 /** Which animation frame is currently showing? */
43
44 /**
163     if (timer <= 0) {
164         timerExpired();
165         startTimer();
166     }
167 }
168
169 /**
176 }
177
178 /**
205 }
206

```

```

207  /**
212  }
213
214  /**
220  }
221  }

```

Listing 8.14: Swimmer: All about timers

The timer field is **private**; the methods for manipulating the value are all **protected** because it is assumed that classes extending `Swimmer` will treat those as *extension points*, spots where the child class can modify its behavior relative to the parent class.

The key concept is that the method `getTimerCountdownTime` is used to get the amount of time the timer will take before expiring; it is called from `startTimer` whenever the timer needs to be restarted. Note that `getTimerCountdownTime` in `Swimmer` just returns a constant value; it is possible to use any amount of calculation or state information to determine the timing (we will see this in `FastSwimmer` where the timer determines how long the swimmer waits off screen or, if the timer is on the screen, how long each animation frame is).

When the timer is decremented (the call to `decrementTimer` is inside `advance` for `Swimmer` and should be in that method for all subclasses of `Swimmer`), a check is made to see if the timer has crossed 0.0 (has it expired). If the timer expires, calls are made to `timerExpired` and `startTimer` to restart the timer. Since the state may have changed, the call to `getTimerCountdownTime` may return something different.

In `Swimmer`, in line 213, `timerExpired` calls `setFrame`. What is the value of the parameter expression passed to `setFrame`? `frame` and `FRAME_COUNT` are both `int` fields, one **static** (Which one? Notice how even capitalization conventions, consistently applied, improve readability of expression). So, `frame + 1` is an integer one greater than `frame`.

What is the `%` operator again? It is the *modulus* or remainder operator. Thus, if `FRAME_COUNT` is set to 4, the result of modulus will be an integer on the range [0-4). Thus `frame` is cyclic, taking on the values 0, 1, 2, 3, 0, 1, ... and so on. This frame number is how we know how to change the appearance of the `Swimmer` sprite.

```

187  if (frame == 0) {
188      rotate(20);
189  } else if (frame == 1) {
190      rotate(20);
191  } else if (frame == 2) {
192      rotate(-20);
193  } else if (frame == 3) {
194      rotate(-20);
195  }
196  }
197
198  /**

```

Listing 8.15: Swimmer: setFrame

The code for `setFrame` shows that the swimmer is rotated relative to its current facing. Assuming frame 0 is the nominal position, when advancing to frame 1 the swimmer rotates 20 degrees clockwise. With frame 2, the swimmer returns to the nominal position (rotates 20 degrees counter-clockwise). Then, advancing to frame 3, the swimmer rotates 20 degrees counter-clockwise from the nominal position. Then, when frame goes to 0 again, the sprite is returned to the nominal position.

This is a very simple state machine where the frame number records the current state and `setFrame` updates the frame number and the appearance of the sprite. Note that this cyclic state is maintained “automatically” by calling `decrementTimer` in `advance`.

Using timer's Extension Points

How can we use the timer in `FastSwimmer`? There is one requirement put on us to use it: `FastSwimmer.advance` must call `decrementTimer`, either directly or by delegating some or all of its work to `Swimmer.advance` through a call to `super.advance`.

We cannot call `Swimmer.advance`. The `static` method `bumpTheWall` is called when a swimmer, using `Swimmer`'s `advance` method hits a wall. The `FastSwimmer` hitting the wall should not cause the grid of swimmers to change direction and move closer to the bottom of the screen. To avoid this, `FastSwimmer.advance` does all its work locally:

```

42     if (isSwimming()) {
43         translate(dT * getDeltaX(), dT * getDeltaY());
44     }
45 }
46
47 /**

```

Listing 8.16: `FastSwimmer`: `advance`

It calls `decrementTimer` and then, if the fast swimmer is on the screen and swimming, it translates the location according to the current velocity. This is a case of repeating ourselves: why can't we reuse the `advance` method declared in `SpriteWithVelocity`? Wasn't avoiding just this sort of code in *all* the sprites in this game the point of factoring out the super class?

Short answer: Yes. The point was to reuse implementation. Unfortunately there is no Java notation for talking about the definition of a method in the class above the class above the current class. That statement tells us what `super`, used with a dot, actually does: the expression following the dot is evaluated as if it had been written in the body of the class above the current one. Typically that means that it calls a method defined in the class the current class extends. But if the closest definition of a named method is further up the hierarchy, `super` will refer to that definition. It always refers to the lowest definition that happens *above* the current class.

Because we don't want `Swimmer.advance` and there is no way from `FastSwimmer` to specify `SpriteWithVelocity.advance`, we were reduced to copying the method body into our `advance`⁶. We are lucky in this case that there is a single line to copy; if the processing were more complex it would make sense to move the actual movement code out to its own protected method so that it could be called from `SpriteWithVelocity.advance` and any other `advance` where it was needed.

8.5 Finishing Rescue Mission

This section will finish `RescueMission` by finishing a discussion of the code of `FastSwimmer` and then look at how multiple levels are supported in FANG.

Launching a FastSwimmer

There is one `FastSwimmer` in the game; sometimes it is swimming across the screen and sometimes it is waiting. This is an object with two states. When constructed, a `FastSwimmer` is waiting; it changes from swimming to waiting when it reaches the edge of the screen and `bounceOffEdges` is executed.

```

169     super.timerExpired();
170 } else {
171     launch();
172 }
173 }

```

⁶ Some programming languages, such as C++, support picking and choosing which ancestor class's implementation to call; that power comes at the cost of having to know more of the class hierarchy.

174 }

Listing 8.17: FastSwimmer: timerExpired

The `timerExpired` method (as mentioned above) launches the fast swimmer. When it begins waiting the `FastSwimmer` sets a random waiting time on the range `[0-MAX_TIME_OFF_SCREEN)` seconds. When the timer expires, since the state is waiting (`isWaiting` returns `true`), `isExpired` calls `launch`. If the fast swimmer is already swimming, the timer is used to animate the swimmer by calling `super.timerExpired` and reusing the animation code written for `Swimmer`.

```

117     if (Game.getCurrentGame().randomDouble() < 0.5) {
118         launchLeftToRight();
119     } else {
120         launchRightToLeft();
121     }
122 }
123
124 /**
131     setRotation(90);
132     setVelocity(speedX, speedY);
133 }
134
135 /**
142     setRotation(-90);
143     setVelocity(-speedX, speedY);
144 }
145
146 /**

```

Listing 8.18: FastSwimmer: launch...

When a fast swimmer launches the state is changed to swimming (`startSwimming` is called). This changes the state, makes the sprite visible, and sets the frame number to 0. Then `launch` “flips a coin” or rather gets a random number from the `Game`. If the number is less than half, the fast swimmer moves left-to-right; otherwise it moves right-to-left. The two methods for actually positioning the swimmer and setting the velocity do what you would expect.

Naming methods well is an important skill. It would be possible to save 10-25 lines by moving the bodies of `launchLeftToRight()` and `launchRightToLeft()` directly into `launch`. Each call would be replaced by three lines of code. It would greatly decrease the readability of `launch` and additional comments would be necessary to explain the code.

The current structure separates the decision about what direction the swimmer should move from how it is set to move in a given direction. The separation of functionality into small, independent units, makes life easier for the programmer. Each method should do one thing and should have a name that reflects the one thing that it does. If you’re having trouble coming up with a good name for a function, think carefully about whether it really only does one thing.

In the current structure it would be easy to have newly created `FastSwimmers` always start moving left-to-right. It would also be easy to change the chances that the swimmer goes in either direction without having to worry whether any code for launching needed to change.

The Rescuer and the Ring

How are swimmers rescued? That is, how do the `Rescuer` moving across the bottom of the screen (a lot like the paddle in `SoloPong` in Chapter 5). It reuses the `advance` method of `SpriteWithVelocity`.

```

44     if (curr.keyPressed()) {
45         char key = curr.getKeyPressed();

```

```

46     if ((key == KeyEvent.VK_LEFT) || (key == KeyEvent.VK_KP_LEFT)) {
47         setVelocity(-Math.abs(getDeltaX()), getDeltaY());
48         super.advance(dT);
49     } else if ((key == KeyEvent.VK_RIGHT) ||
50         (key == KeyEvent.VK_KP_RIGHT)) {
51         setVelocity(Math.abs(getDeltaX()), getDeltaY());
52         super.advance(dT);
53     } else if (key == KeyEvent.VK_SPACE) {
54         ring.startFlying();
55     }
56 }
57 }
58
59 /**

```

Listing 8.19: Rescuer: advance

The resetting of the x-component of the velocity (lines 49 and 53) use the `Math` class, a class which has only **static** methods. `abs` is the absolute value or the magnitude of the number without regard to sign. Thus it is always a non-negative number. Since we don't know whether the rescuer was last moving from left-to-right or reverse that, we convert the x-component to a positive value (it must be non-zero for the rescuer to move) and the invert it if moving to the left.

If the user presses the space bar, the rescue ring is launched. The rescue ring, like the `FastSwimmer` is a sprite with two states: ready and flying. When the ring is ready, it is “attached” to the rescuer, matching location with the rescuer each frame.

```

100     super.advance(dT);
101     rescueRope.setStart(rescuer.getLocation());
102     rescueRope.setEnd(getLocation());
103 } else {
104     setLocation(rescuer.getLocation());
105 }
106 }
107
108 /**

```

Listing 8.20: RescueRing: advance

If the ring is ready it is not flying. Thus line 106 executes, moving the rescue ring to the location occupied by the rescuer. The first branch of the `if` statement uses the `super` definition of `advance` to move the ring.

Then a line is drawn between the rescuer and the rescue ring (or rather, the line sprite `rescueRope` has its end points set to be the location of the rescuer and the location of the rescue ring). The rope is purely aesthetic; no other sprites ever interact with it.

Changing the state of a `RescueRing` takes place in `startFlying` and `startReady`: `startFlying` is called in `Rescuer.advance` (when space bar is pressed); `startReady` is called from the constructor, `bounceOffEdges`, and `isRescuing`. The rescue ring keeps flying until it rescues a swimmer (regular or fast) or it hits an edge of the screen. With this feature the game values accuracy very highly.

Rhythm in Games

How the `RescueRing` behaves is a central design decision for this game. How many rings can the rescuer throw at one time? What is the “recharge time” between rings? The decision for `RescueMission` was that there is only one ring and once it is thrown it travels in a straight line until it rescues a swimmer or moves off the top of the screen.

This decision means that pressing the space bar will call `startFlying` but that method will do nothing unless the ring is currently ready; if the ring is already flying, `startFlying` does nothing. This means there

is only one ring in flight. The ring is returned to the ready state when it rescues a swimmer or goes off the screen so the limitation in `startFlying` enforces just the design described above.

This design decision forces the player to trade off their long-range accuracy (say for hitting the fast swimmer) and the need to throw the rescue ring often. This particular mechanic was lifted from the retro game on which `RescueMission` was modeled, *Space Invaders*. In Taito's arcade *Space Invaders*⁷ there was an additional rain of missiles from the grid of invaders and a series of destructible forts between the invaders and the player. There was no good story reason for the swimmers to want to harm the rescuer (and the game is already almost 1400 lines long) so that was left out.

```

114     startReady();
115     }
116 }
117
118 /**
195     readyForLaunch = false;
196     rescueRope.show();
197     rescueRope.setStart(rescuer.getLocation());
198     rescueRope.setEnd(getLocation());
199     }
200 }
201
202 /**
207     readyForLaunch = true;
208     rescueRope.hide();
209     setLocation(rescuer.getLocation());
210     }
211 }
212 }

```

Listing 8.21: `RescueRing`: States and Bouncing

Internally, `RescueRing` uses a `boolean` field, `readyForLaunch` to keep track of its state. When the field is `true`, the ring is in the ready state; when the field is `false`, the ring is in the flying state. Lines 197 and 209 take care of setting the field according to the state the ring is changing to.

When the ring is flying, the rescue rope should be visible; when it is ready, the rope should not be visible. Finally, each state setting method adjusts the location of either the ring or the rope, depending on what the new state is.

Bouncing off the edges is running into a wall. When that happens, the flying ring goes from flying to ready (line 116 above).

```

160     }
161
162     /**

```

Listing 8.22: `RescueRing`: `isRescuing`

How does a ring rescue a swimmer? The action needed to rescue the swimmer is actually beyond the scope of `RescueRing`; it is something that the *game* is in charge of. The ring does have a `boolean` method, `isRescuing` which takes a swimmer as a parameter and returns `true` if the swimmer and the ring are intersecting.

Note that this method uses short-circuit Boolean evaluation to make sure that `intersects` is not called with `null`. The `null` check is necessary because of how we decided to handle rescued swimmers: they are set to `null` when rescued. By putting the check here we can write a simple iteration to handle checking each and every swimmer to see if it has been rescued.

The iteration is in `RescueMission.rescueIfPossible`.

⁷*Space Invaders*, released in 1978, remains trademarked by Taito

```

273     for (int row = 0; row != ROWS; ++row) {
274         for (int column = 0; column != COLUMNS; ++column) {
275             Swimmer curr = swimmers.get(row).get(column);
276             if (ring.isRescuing(curr)) {
277                 rescueSwimmer(row, column);
278             }
279         }
280     }
281 }
282 if (fastSwimmer.isSwimming() && ring.isRescuing(fastSwimmer)) {
283     rescueFastSwimmer();
284 }
285 }
286
287 /**

```

Listing 8.23: RescueMission: rescueIfPossible

The nested loops look similar to those used in `moveEverything`; the only difference is that rather than checking against `null` the `if` statement checks if the ring is rescuing the given swimmer. Remember that there is a `null` test inside of `isRescuing` so this is safe. If the check returns `true`, the method calls `rescueSwimmer`.

After the loop, the fast swimmer is checked to see if it is being rescued. It is necessary to check `isSwimming` on the fast swimmer because it cannot be rescued if it isn't visible. It might be possible to intersect with it even when it is off the screen (the reason we remove `Swimmer` sprites from the list of lists to rescue them).

```

264     fastSwimmer.startWaiting();
265     ring.startReady();
266 }
267
268 /**
269     setScore(getScore() + curr.getScore());
270     removeSprite(curr);
271     swimmers.get(row).set(column, null);
272     ring.startReady();
273 }
274
275 /**

```

Listing 8.24: RescueMission: rescue...

To rescue the fast swimmer we add the fast swimmer's score to the game score and reset the state of the fast swimmer (to waiting) and the rescue ring (to ready). To rescue a regular swimmer we do just the same thing but regular swimmers don't have a waiting state. Instead we set the location where the sprite is in the grid to `null`, see line 301, and remove the sprite from the FANG data structure so it is no longer painted as part of each frame, see line 300.

Levels in FANG

The concept of Game in FANG is a little more complicated than this text has let on. Internally FANG keeps a list of Game derived class objects. The first item in the list is the current game (and is what is returned by the call `Game.getCurrentGame`). When the current game really finishes, FANG automatically calls the next game in the list. If there is no next game, then FANG terminates the program.

Each game in the list can be considered a level. Thus when a level is over (when all swimmers in the grid are rescued) we can create a new `RescueMission`, add it to the list of games FANG maintains, and then really end the current game. That is exactly what happens in advance:

```

104     bounceEverything();
105     rescueIfPossible();
106     if (checkIsLevelOver()) {
107         addGame(new RescueMission(level + 1, getScore(),
108             swimmerDX * LEVEL_SPEEDUP, swimmerDY * LEVEL_SPEEDUP));
109         finishGame();
110     } else if (checkIsGameOver()) {
111         addGame(new EndOfGame(getScore()));
112         finishGame();
113     }
114 }
115
116 /**

```

Listing 8.25: RescueMission: advance

Everything is moved, then everything is bounced off the edges of the screen and every swimmer is checked for a rescue. Then if the level is over (there are no remaining swimmers in the grid), a new game is added by calling `addGame` with a newly constructed game. The new game is constructed with a level number, the current score, and slightly higher speeds for the swimmers.

If those four parameters are necessary for constructing a `RescueMission`, how does FANG know what values to use for the first level? The short answer is that it doesn't. FANG requires every `Game` derived class to have a default constructor; the default constructor is the no parameter constructor. When creating the first level, FANG calls the default constructor.

```

75 }
76
77 /**
78     this.scoreSprite = null;
79     this.level = level;
80     this.swimmerDX = swimmerDX;
81     this.swimmerDY = swimmerDY;
82     this.initialScore = initialScore;
83 }
84
85 // Methods -----

```

Listing 8.26: RescueMission: Constructors

Applying the DRY principle, we just forward the default constructor to the four parameter constructor providing initial values: level 1, score 0, and the named constants for swimmer velocity.

The only odd thing remaining in `advance` is line 113. What is `EndOfGame`? It is used with `new` so it must be a class. Is it part of FANG? Looking at the documentation the answer is no. It is a fairly simple “game”, one which takes a score as a parameter to its constructor and displaying a `StringSprite` on the screen. It is like the drawings and examples in Chapter 2: it has a setup method but no advance.

```

1 import fang.core.Game;
2 import fang.sprites.StringSprite;
3
4 /**
5  * The "game" is really just a level of the main game. It is constructed
6  * to display a message on how well the player did (it is initialized
7  * with their score) and then the setup creates a centered end-of-game
8  * message.
9  */

```

```

10 public class EndOfGame
11     extends Game {
12     /** the score provided to the constructor */
13     private final int initialScore;
14
15     /**
16      * Construct a new EndOfGame level. Requires the score to display for
17      * the player.
18      *
19      * @param initialScore the player's score
20      */
21     public EndOfGame(int initialScore) {
22         this.initialScore = initialScore;
23     }
24
25     /**
26      * Setup the display information on the screen.
27      */
28     @Override
29     public void setup() {
30         setScore(initialScore);
31         StringSprite announcement = new StringSprite();
32         announcement.setText("GAME OVER\nFinal Score: " + getScore());
33         announcement.setScale(1.0);
34         announcement.setLocation(0.5, 0.5);
35         addSprite(announcement);
36     }
37 }

```

Listing 8.27: EndOfGame: Whole Class

This use of a level to mark the end of the game is similar to how many video games actually work. They have modes of operation such as attract mode (show pretty video or pictures to get players interested), setup (setting up the game), the lobby (waiting to play multiplayer games), the game itself, loading screens, and high-score or end-of-game levels. Note that it would be fairly simple to add a small advance to `EndOfGame` that would start a new game of `RescueMission` when the player pressed the space bar.

8.6 Summary

Nested Loops

The body of a `for` loop can contain any valid Java code. This means that it can, by definition, include another `for` loop. A collection of loops, one inside another, are *nested loops*. To determine how many times the body of the inner-most loop executes, it is necessary to determine how many times the body of each loop executed and then *multiply* the values together. This is simple for count-controlled loops that run a fixed number of times; it is more difficult if termination conditions of the loops are more complex.

When working with a two-dimensional structure like a multiplication table, it is useful to think of using two nested loops (one loop for rows, an inner loop for columns).

Collections of Collections

While nested loops permit generating two-, three-, and higher-dimensional things, keeping the data for such a structure typically requires a list of lists (of lists of lists of ...). That is, where a loop per dimension permits

discussing a multidimensional creation like a multiplication table, a collection per dimension permits keeping the structure around.

An `ArrayList` of `ArrayList`s of some object gives a two-dimensional data structure which can hold a “table” or grid of the element objects. To fill such a structure requires two nested loops (one for rows and one for columns) and $rows \times columns + rows + 1$ calls to `new`: one for each element in the table or $rows \times columns$ calls for those, one for each row or $rows$ calls for those, and one call for the whole list of lists.

Traversal of a data structure means passing over all of the elements in the structure. For a list of lists, this implies a pair of nested loops. In this chapter we saw how to count elements of a list of lists with a given attribute (not being `null` in our case) and how to find the first element with a given attribute (again, not being `null` though the `if` statement could be modified fairly simply).

Because `ArrayList`s can only hold references, all plain old data types have object equivalents: `Integer` for `int`, `Double` for `double` and `Boolean` for `boolean`.

Inheritance

An oak *is-a* tree just as a cherry tree *is-a* tree. This relationship means that any time a generic `tree` is required, either an `oak` or `cherry tree` would be acceptable as a `tree`.

In Java, an object of any class which extends another (either directly or through a chain of intermediate classes), *is-a* object of the ancestor class as well. Thus a `CompositeSprite` *is-a* `Sprite`, a `RescueMission` *is-a* `Game`, and a `Rescuer` *is-a* `SpriteWithVelocity`, *is-a* `CompositeSprite`, *is-a* `Sprite`, and *is-a* `Object`.

Animation

Animation, the changing of the appearance of a `Sprite`, is similar to regular movement of a sprite on the screen. The only difference is if the frequency of the updates should be different than calls to advance. This chapter demonstrated how to use a timer and how to include **protected extension points**, methods which can be overridden in subclasses to extend or modify the behavior of the timer in specific subclasses.

Combining overridden methods and the *is-a* relationship yields the important object oriented programming principle of *polymorphism*. Polymorphism is having many forms; the superclass provides an interface that can be used in the general case (the timer expires) and the subclasses provide specialized implementations of the extension point routines which are called through the superclass reference.

Levels in FANG

The `Game` class in FANG has an `addGame` method which adds a `Game`-derived class to the list of games to be run by the current game. When a running game calls `finishGame`, the currently running game is taken off the front of the list of games and if there is any game remaining in the list, the first is started as the next game or next level.

The chapter makes use of this by constructing increasingly more difficult levels of the same game when all the swimmers are rescued. It also uses this fact to create a special end-of-game level which displays a message for the player showing their score. This is where a level could be built to read/save high scores or the game could be started over.

Java Templates

Chapter Review Exercises

Review Exercise 8.1 Reimplement TicTacToe with 2D array

Review Exercise 8.2 Battleship!

Programming Problems

Programming Problem 8.1 Make the game play again from the beginning

Programming Problem 8.2 Have different messages for different levels

Programming Problem 8.3 Different scores for fast swimmers

Scanner and String: Character Input

So far all of our games were *self-contained* and *stateless*. They are self-contained in the sense that they don't rely on any files other than their Java source code and then, after being compiled, the compiled `.class` files. This is beneficial in that the games are simple: distributing the game so someone else can play it depends only on passing along the `.class` file. The other side of the self-contained coin is that a game *must* be recompiled to change how it works. This is not how most programs work. You do not to recompile **Microsoft Word**[®] each time you want to edit a different file. You don't have to recompile the `javac` executable each time you want to compile a different program. This is because these programs interact with the *file system* on the computer where they run. The only instance where we have interacted with the file system is when loading a `ImageSprite`.

Our games are stateless in the sense that they do not have any sense of whether this is the first, the one hundredth, or the one millionth time they have run. Each time the program starts it starts from exactly the same place. There may be random numbers involved to change the play from run to run (think `NewtonsApple`: starting location of the apple is random but the sprites on the screen are fixed at compile time. This is related to being self-contained but state would mean that the running game could leave some record of what it has done. Consider keeping track of a high score list or saving a game in progress so that it can be restored.

This chapter will address the self-contained half of the problem by introducing `Files`, a way to connect to and read files on the file system. This means that the game can change its performance without being recompiled; consider games with multiple levels, user-designed content, and mods. We defer reading information from disk files until Chapter 11.

9.1 Designing Hangman

Imagine a two-player game where one player, the chooser, selects a word and exposes the number of letters to the other player, the guesser. The guesser guesses letters, one at a time, and the game is over when the guessing player guesses all of the letters in the word or when they miss six times. This game could be played with paper and a series of tick marks; when the sixth tick is written, the guesser loses and the chooser exposes the rest of the word.

This is an example of a simple word game with simple rules: pick a word, guess letters, count misses, mark matches, guess word, or miss too many letters. It lacks a good story, though. The description is too dry.

Enter *Hangman*: instead of tick marks, the chooser draws a gallows at the beginning of the game and on each missed letter draws a new body part hanging from the gallows; when the last body part is drawn, the guesser is “dead”. This is an example where a story (perhaps a bit morbid) can add interest to an otherwise dry game design. This particular game has been around for more than a century[?].

Our Hangman game will pit the human player, as the guesser, against the computer playing as chooser. The design diagram shows that the screen has four sprites on it: the score, the gallows, the word being guessed, and the alphabet selector.

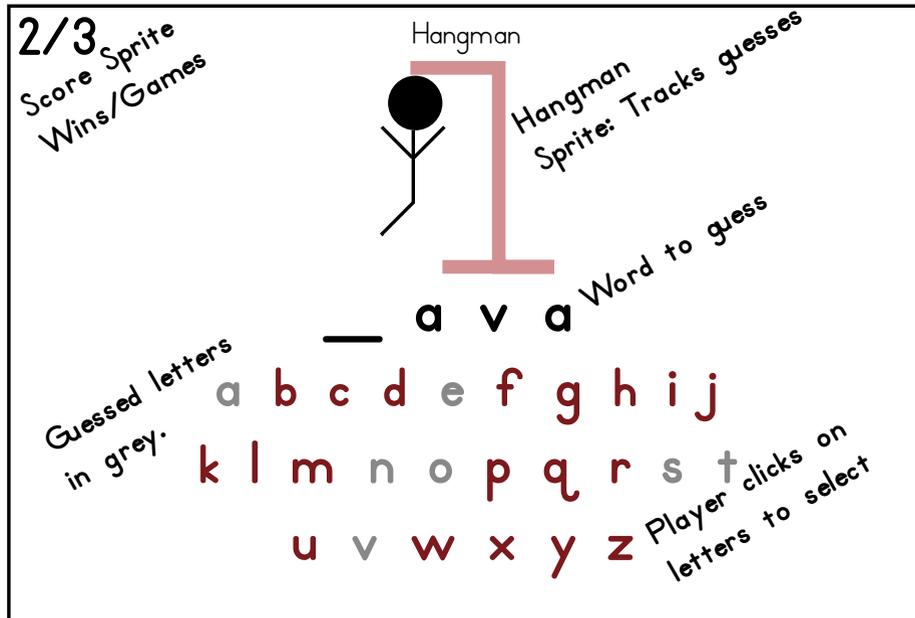


Figure 9.1: Hangman game design diagram

Looking at the diagram, there seem to be a lot more than four sprites on the screen: the alphabet is 26 characters and the hangman/gallows has at least 6 parts. Top-down design means that we will use *abstraction* to conquer the complexity in this program. We will design the interfaces of the four classes we use in the game and then we will implement the game using those interfaces. We will discuss implementing the four sprite classes at the end of the chapter because all of the new concepts in this chapter are part of the game proper.

Public Interfaces

ScoreSprite

The `ScoreSprite` in the upper-left corner of the game implements a scoring model just like that found in `NewtonApple` back in Chapter 5. It is just wrapped up in a class that knows how to initialize, how to record a win, and how to record a loss. That means the public interface for the class is:

```
public class ScoreSprite
  extends StringSprite {
  public ScoreSprite()...
  public ScoreSprite(int wins, int games)...
  public void lose()...
  public void win()...
}
```

The sprite is added to the game, properly positioned. Then, whenever the game is won or lost, `win` or `lose` is called and the `ScoreSprite` will fix up its score¹ and update the displayed value. The overloaded constructors provide flexibility: the sprite can be initialized to any arbitrary value. The number of wins cannot be greater than the number of games played.

¹Going back and fitting the `ScoreSprite` into `NewtonApple` is left as an exercise for the interested reader.

HangmanSprite

The HangmanSprite is a composite of the gallows and the victim. Note that the hangman takes the place of the tick marks so it is really a second score indicator. It shows the score for the current game while the ScoreSprite shows the scores across multiple games.

Thinking about it as a score indicator gives us the insight to build the public interface:

```
public class HangmanSprite
    extends CompositeSprite {
    public HangmanSprite()...
    public void clear()...
    public void incrementState()...
    public void setColor(Color color)...
    public boolean stillKicking()...
}
```

The local `setColor` method sets the color of the victim's body to the new color. `clear` clears the state so no body parts are on display (sets the score to 0). `incrementState` adds one to the state and displays a new body part. `stillKicking` returns `true` until the victim is finished being hung. They are finished when all the body parts are visible.

GuessableWord

The word to be guessed is displayed in the middle of the screen. It is a `StringSprite` with additional state. It tracks the word to be guessed and the progress of the guess (it shows “_” characters where unguessed letters are and the letters where they have been guessed). The public interface is:

```
public class GuessableWord
    extends StringSprite {
    public GuessableSprite(String word)...
    public void expose()...
    public boolean guess(char letter)...
    public boolean isGuessed()...
}
```

A `GuessableWord` is initialized with the word to be guessed. As part of the implementation it will prepare the displayed version of the word. Then, so long as `!isGuessed` the game goes on. Each time the user picks a letter, the letter will be passed to `guess`: if the letter is in the word it is exposed and `guess` returns `true`; if the letter is not in the word then the displayed word is unchanged and `guess` returns `false`.

AlphabetSelector

The user must be able to pick letters to guess. This could be done with the keyboard, with the mouse, or with both. For simplicity, we will focus on just using the mouse (extending the sprite to handle keyboard input is presented as an exercise).

An additional problem in Hangman is that the guesser would like to keep track of letters already guessed; if the letter is *in* the word it is easy to remember but if it isn't in the word it is not clear how they are remembered. The `AlphabetSelector` will contain `LetterSprites`, sprites which have a ready and a used state; they will be shown in two different colors and only ready letters will be selectable by the player. Thus the appearance will give feedback to the user that they shouldn't reselect letters and the sprite will also keep them from making the mistake.

```
class AlphabetSelector
    extends CompositeSprite {
    public AlphabetSelector()...
    public char selectedChar()...
```

```

    public void unselectAll()...
}

class LetterSprite
    extends StringSprite {
    public LetterSprite(char letter)...
    public char getLetter()...
    public Color getReadyColor()...
    public Color getUsedColor()...
    public boolean isReady()...
    public boolean isUsed()...
    public void setReadyColor(Color readyColor)...
    public void setUsedColor(Color usedColor)...
    public void startReady()...
    public void startUsed()...
}

```

The public interface of `LetterSprite` is included to make the operation of `AlphabetSelector` clearer. When an `AlphabetSelector` is built, it contains 26 `LetterSprites`, all initialized to the ready state. `selectedChar` returns either a letter (if one was selected) and the *null character*, the `char` with a value of zero, otherwise. If a letter was selected, the corresponding `LetterSprite` is changed from the ready to the used state.

Once a `LetterSprite` is in the used state, it cannot be selected again. To permit restarting with the whole alphabet selectable again, the `unselectAll` method is provided.

Hangman: The Game

Given these classes, since they do most of the work, designing Hangman is easy. The setup routine is broken into four subroutines, one for each of the sprites discussed above. They are created, scaled, located, and colored according to the design diagram shown above. We will skip covering the game's setup method in detail because it is so similar to code we have already covered.

```

38     char ch = alphabet.selectedChar();
39     if (ch != '\0') {
40         if (word.guess(ch)) {
41             if (word.isGuessed()) {
42                 scoreSprite.win();
43                 doneWithGame("Congratulations");
44             }
45         } else {
46             hangman.incrementState();
47             if (!hangman.stillKicking()) {
48                 word.expose();
49                 word.setColor(getColor("red"));
50                 scoreSprite.lose();
51                 doneWithGame("You lose!");
52             }
53         }
54     }
55     } else {
56         if (this.getKeyPressed() == ' ') {
57             setGameOver(false);
58             startOver();
59         }
60     }

```

```

61 }
62
63 /**

```

Listing 9.1: Hangman: advance

Hangman uses a single level (and level type), unlike Chapter 8’s `RescueMission`. Instead Hangman has two different states: game over and not game over.

Style note. Notice that the `if` statement is written without using the `!` symbol. Software engineering TK.

If the game is already over, the game is waiting for the user to press the space bar to restart the game. The game was ended with a call to `endGameWithMessage` (definition given below) which gives displays a specific message about how the game ended and also shows a message to press the space bar to start again. Thus the player knows what to do and the game waits for them to do it. `startOver` is a FANG routine which restarts the current Game; it is almost like adding the current game to the list of games and then calling `finishGame`.

While the game is going on the `else` clause of the `if` statement is executed. Line 46 checks if a letter (or the null character) has been picked. If the character is not the null character, then the player has made a guess.

The guess is passed into the `guess` method of `GuessableWord`; as it is described above, `guess` returns `true` if the letter was found in the hidden word. Thus if it is `true` it is possible that the player has won the game. If they have, `endGameWithMessage` is called with a win message and the score is incremented.

If `guess` returns `false` then a new body part is added to the gallows and a check is made to see if the player has been hung. If the player has been hung, the game is ended with an appropriate message and the score is updated for the loss. It is also necessary to expose the word so the player knows the word that they missed.

```

72     "\nPress <space> to play again.");
73     restartMessage.setScale(0.9);
74     restartMessage.setLocation(0.5, 0.75);
75     addSprite(restartMessage);
76     alphabet.hide();
77     setGameOver(true);
78 }
79
80 /**

```

Listing 9.2: Hangman: endGameWithMessage

The `endGameWithMessage` method creates a new `StringSprite` containing two lines: the given message and a game restart line. The sprite is positioned in the bottom half of the screen, the `AlphabetSelector` is hidden, and the game is set to over.

These two methods are the only two needed for playing the game. The rules for picking a letter, for guessing a letter in a hidden word, and for keeping track of scores are abstracted away from the game program itself; all it has to do is coordinate communication between the pieces that know how to play the game.

Stub Methods

We will take a look at one of the setup methods because it leads us to a way of applying top-down design, the idea of *stub methods*. A stub method is a temporary implementation of a method, a simple implementation which permits compiling and testing of higher level methods. A stub method is an implementation of the design admonition, “Pretend that it works.”

```

132     word.bottomJustify();
133     word.setWidth(0.7);
134     word.setLocation(0.5, 0.6);
135     addSprite(word);
136 }

```

```

137
138  /**
145     return "cats and dogs";
146  }
147  }

```

Listing 9.3: HangmanWithStub: setupWord and wordLoad

At line 133 `setupWord` calls `wordLoad`². Lines 145-148 represent a stub implementation of `wordLoad`. The method just returns a fixed word. Every time the game is played, the chooser will choose the exact same word.

This does not make for a good Hangman game but it does make for a *playable* and (more importantly) *testable* Hangman game. With the stub implementation for `wordLoad` in place we can compile `HangmanWithStub` and all of the supporting classes without knowing anything about opening files for input. Thus we can debug the mechanics of the game separately from the problems of locating a file full of possible words, reading that file into memory, and picking a word from that list. As always, separating problems into different levels either by delegation (as we have done here) or abstraction (as partitioning `Hangman` into the four specialized sprite classes), helps us overcome complexity.

We will take a slight detour now to discuss how `java` runs a compiled `.class` file from the command-line, how we can specify parameters to a program when it is run, and how `FANG` has started the program for you automatically behind the scenes.

9.2 Starting Programs

Remember the introduction of *operating systems* in Section 2.2? An operating system is a program which can start and stop *other* computer programs. One of the biggest differences between operating systems are the rules the operating system uses when beginning the execution of a program. An executable program that runs directly on a given CPU requires more than just the right CPU: it also requires the right operating system so that it is properly loaded into memory and the correct *entry point* is chosen to begin execution.

An example of this difference lies in the executable formats for *Microsoft Windows*, *Linux*, and *Apple OSX*. All three operating systems run on computers with an Intel or compatible CPU; all three operating systems use sequences of instructions drawn from the exact same instruction set as do programs which any of the operating systems can start.

Without special adaptation, however, programs compiled on OSX cannot be started directly on a Windows computer, nor can Windows executables be started directly by Linux, and so on. The instructions in the program depend on the CPU but how the program starts depends on the CPU and the operating system.

This is of interest to us because we are programming using the Java programming language. Rather than compiling to machine code run directly on any given CPU (or loaded by any given operating system), the `javac` compiler compiles to `.class` files containing *bytecodes* to be interpreted by the `java` virtual machine program.

The `java` interpreter is like an operating system in that it has a defined convention for how it starts a program. When you run a given `.class` file, `java` calls a specific, `static` method, the method with the following signature:

```
public static void main(String[] args)
```

We will address the meaning of the two square brackets later in this section. When `java` is run with the command-line:

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar Hangman
```

the Java interpreter sets up all appropriate library connections, loads the `Hangman` class, and then calls `Hangman.main` with the previous signature.

The complete *lifecycle* of an application is:

² `wordLoad` might not be the greatest method name but it has the convenient property that when the source code is automatically sorted by method name, it sorts to the end of the class. This means that through line 145 `Hangman.java` and `HangmanWithStub.java` are identical. The reader is urged to consider what better names for this method might be.

- Load the named `.class` file
- Call `main` in loaded class
- Terminate program when `main` returns

“Wait,” you are thinking, “No where in this book have we written a `main` method. Looking at `Hangman.java` (and `HangmanWithStub.java`), there is no such method.” FANG uses the standard Java inheritance model to provide a version of `main` for you. In `GameLoop.class`, there is a **static** `main` method with the right signature.

When Java runs an *application*, it calls `main` as defined by the class named on the command-line calling the interpreter. In FANG the default implementation of `main` is both the simplest and the most complex method in the library. It is simple in *what* it does: it determines the class which was specified on the command-line for Java, makes sure the class extends `GameLoop`, calls the default constructor for that class, and then calls `runAsApplication`. It is complicated in that figuring out what class was specified when the program was run and then calling the constructor for a type that is unknown at compile time requires some deep knowledge of Java.

We will stick with the simple definition, ignoring the convoluted details. That does leave the question: What does `runAsApplication` do?

Applets v/ Applications

One powerful feature of FANG is its ability to run exactly the same code as either an application (with a call to `java` from the command-line) or as an *applet* inside of a Java-enabled Web browser.

What is an applet and how is it different from an application? An applet is a class which extends `java.applet.Applet`, a Java library class. The lifecycle of an applet is:

- Load named `.class` file
- Construct (with default constructor) an object of the given class
- Call `init` on the new object
- While browser is running:
 - Call `start` when page containing applet is entered
 - Call `stop` when page containing applet is exited
- Call `destroy` on the applet object

Because the applet runs inside the context of a Web browser³, it is not started with a call to `main` because the Java virtual machine has already started and is running the browser’s version of `main`.

What does that mean for a FANG game? A FANG game must run as either an applet or an application and it must *not* require recompiling to change from one to the other. If FANG provides its own `main` and that `main` acts like an applet browser, implementing the applet lifecycle outlined above, all started from a call to `main`, then a FANG program can run like an applet no matter how it is started.

Notice that this is another separation of function, the splitting of a problem into component parts that are solved independently. Rather than worry about “How to start and run a program as an applet or an application”, FANG solves the problems of “How to run a program as an applet” and “How to provide a `main` which starts an applet running inside of an application.”⁴

When a program is run from the command-line, it is possible to provide parameters. These parameters are just like parameters passed to any method except that they are each `Strings`. The next section discusses how to interpret the arguments passed in to `main` both in that method, when you write your own, and how FANG saves those parameters for you so that you can have access to them without having to override `main` yourself. This information is important because we want to be able to write our own programs without the FANG library by the end of the book.

³or another applet browser such as `appletviewer` in the JDK

⁴One of the authors (Ladd) originally wrote a different toolkit with an inherited `main` method in summer, 2006, as part of an early version of this book. The current implementation of `main` and its support code is based on code written for the ACM’s Java Task Force’s Java toolkit [RBC⁺06]

public static void main(String[] args)

Because of the way Java handles inheritance, any class which extends `GameLoop` can provide its own `main` method with the above signature; such a method overrides the method defined in `GameLoop`. If you want `FANG` to run as usual, your `main` method should end with the following line:

```
InitializeApplication.fangMain(args);
```

You will need to `import` `fang.util.InitializeApplication` in order to call this method. `fangMain` is the wrapper around all of the complexity mentioned above so we will not drill down any deeper into it here.

What can you do in `main`? And what is the meaning of `String[] args`? In reverse order: `String[] args` is an *array* of `String` objects and `main` can do anything that is built into Java.

An array is like a `ArrayList` in that it is a Java collection, a single object containing other objects. An array is unlike an `ArrayList` in its syntax and public interface. To illustrate this and where we are going with writing non-`FANG` programs, the next section will focus on writing a program which prints all of its command-line parameters to standard output. It will *not* use any part of `FANG`.

Writing main

```

1  /**
2  * A program showing how to use a for-loop to iterate across the
3  * arguments passed in to the program when it is run.
4  */
5  public class PrintAllArguments {
6      /**
7       * The main program: java PrintAllArguments starts by calling this
8       * public static method. The method just uses a for-loop and
9       * System.out.println to print each argument on a line by itself.
10     *
11     * @param args The command-line arguments
12     */
13     public static void main(String[] args) {
14         System.out.println("PrintAllArguments:");
15         for (int i = 0; i != args.length; ++i) {
16             System.out.println("  args[" + i + "] = \"" + args[i] + "\"");
17         }
18     }
19 }

```

Listing 9.4: `PrintAllArguments`

This program treats `args` just like any other collection. The following table gathers the differences between an `ArrayList` and an array starting with those shown in this code and including a couple of other cases.

What does this code do? That is, given different command-lines, what is the output?

```

~/Chapter09% java PrintAllArguments
PrintAllArguments:

~/Chapter09% java PrintAllArguments alpha bravo charlie
PrintAllArguments:
  args[0] = "alpha"
  args[1] = "bravo"
  args[2] = "charlie"

~/Chapter09% java PrintAllArguments "alpha bravo charlie"

```

Array	ArrayList	Description
<code>array.length</code>	<code>list.size()</code>	The array exposes a <i>field</i> while the list exposes a method.
<code>array[i]</code>	<code>list.get(i)</code>	The array uses square brackets to indicate individual items.
<code>array[i] = "x"</code>	<code>list.set(i, "x");</code>	The square brackets generate the element location so they can be used to get and set the given element.
<code>array = new String[n]</code>	<code>list = new ArrayList<String>()</code>	The array constructor is called when square brackets containing the number of elements is appended to the element type. The default constructor is called for each of the <i>n</i> element locations as part of construction; while entries can be reset, the number of elements in an array cannot be changed after construction. The list is initially empty; add and remove expand and contract the list as necessary to fit the current number of elements.

```
PrintAllArguments:
  args[0] = "alpha bravo charlie"
```

In the first instance, there are no command-line arguments after the name of the class file for Java to load. Thus line 16 prints the name of the program. The `length` of `args` is 0 so the `for` loop is never executed.

The second instance has three words appearing after the name of the program, `alpha`, `bravo`, and `charlie`. The words are separated from the name of the class and each other by some number of spaces. With three different words, `args.length` is 3 so the loop executes three times and the three `String` values are printed.

The last example shows how the command-line interpreter, provided by the operating system (bash on Linux in this case) can group multiple words together. That is, the command-line arguments after the name of the class are similar to those in the second case, they are just enclosed in double quotes. This causes the command-line interpreter to treat the string enclosed in the quotes is treated as a single argument. `args.length` is 1 and `args[0]` (remember, `[0]` is equivalent to `.get(0)` for an `ArrayList`), the only `String` in the argument array is printed.

Arguments in FANG Programs

The FANG library processes command-line arguments by breaking them into two types: *named* and *unnamed* arguments. An argument of the form `name=value` (where the whole thing is one word according to the command processor) is a named argument. FANG collects all named arguments with the same name and puts the values into an `ArrayList` of `String`, one entry for each time the same name appears. The `ArrayList` is available from the `getNamedArgs(String name)`, a `static` method in `InitializeApplication`.

All other arguments, those without an equal sign, are grouped together as unnamed attributes. They are put into an `ArrayList` which you can access with a call to `getUnnamedArgs()`. `PrintAllArgumentsFANG` demonstrates how unnamed arguments work.

```
1 import java.util.ArrayList;
2
3 import fang.core.Game;
4 import fang.util.InitializeApplication;
5
```

```

6  /**
7   * Uses FANG methods to get the arguments passed into the program.
8   * @author blad
9   */
10 public class PrintAllArgumentsFANG
11     extends Game {
12     /**
13      * List all the command-line arguments on standard output.
14      */
15     @Override
16     public void setup() {
17         ArrayList<String> args = InitializeApplication.getUnnamedArgs();
18         System.out.println("PrintAllArgumentsFANG: ");
19         for (int i = 0; i != args.size(); ++i) {
20             System.out.println("  args[" + i + "] = \"" + args.get(i) + "\"");
21         }
22     }
23 }

```

Listing 9.5: PrintAllArgumentsFANG

```

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
PrintAllArgumentsFANG
PrintAllArgumentsFANG:

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
PrintAllArgumentsFANG alpha bravo charlie
PrintAllArgumentsFANG:
  args[0] = "alpha"
  args[1] = "bravo"
  args[2] = "charlie"

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
PrintAllArgumentsFANG "alpha bravo charlie"
PrintAllArgumentsFANG:
  args[0] = "alpha bravo charlie"

```

(The `\` characters are continuation characters; the lines continue onto the next line. They can be typed on a single line in most command-line interpreters. They are necessary so the lines fit on the printed page.)

The discussion for the three different executions of the program match those given above for `PrintAllArguments`. The real difference between the two programs is that `FANG` does the work of reading the array and puts it all in an `ArrayList` for you.

```

1  import java.util.ArrayList;
2
3  import fang.core.Game;
4  import fang.util.InitializeApplication;
5
6  /**
7   * Uses FANG methods to get the arguments passed into the program.
8   * @author blad
9   */
10 public class NamedArgumentsFANG
11     extends Game {

```

```

12  /**
13   * List all the command-line arguments on standard output.
14   */
15  @Override
16  public void setup() {
17      System.out.println("NamedArgumentsFANG: ");
18      ArrayList<String> names = InitializeApplication.getArgumentNames();
19      for (int i = 0; i != names.size(); ++i) {
20          ArrayList<String> args = InitializeApplication.getNamedArgs(
21              names.get(i));
22          for (int argumentNdx = 0; argumentNdx != args.size();
23              ++argumentNdx) {
24              System.out.println("  args[\"" + names.get(i) + "\"][\" +
25                  argumentNdx + "\"] = \"" + args.get(argumentNdx) + "\"");
26          }
27      }
28  }
29  }

```

Listing 9.6: NamedArgumentsFANG

Named arguments are similar but are grouped by name. The `getArgumentNames` method returns a list of all names used for arguments; this list can be empty. Looking at the code in `NamedArgumentsFANG`, notice that it is a pair of nested loops because the collection of named arguments is a list of lists (a two-dimensional data structure).

```

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    NamedArgumentsFANG one=alpha two=bravo one=charlie two=delta
NamedArgumentsFANG:
  args["two"][0] = "bravo"
  args["two"][1] = "delta"
  args["one"][0] = "alpha"
  args["one"][1] = "charlie"

```

One thing to note is that the order of the strings in the list returned by `getArgumentNames` is, for all intents and purposes, random. This is because of the internal structure used to store the lists of arguments. We will discuss how to sort collections in Chapter ??.

9.3 Different Iteration

So far *iteration* has been synonymous with the `for` loop in Java. This is because we have limited ourselves to *count-controlled* loops. To review, a count-controlled loop is a loop where, at the time the loop begins, the computer knows the number of times the loop will run. We use a `for` loop to implement count-controlled loops because the three parts of the `for` loop, the *<init>*, *<continuation>*, and *<nextStep>*, capture the operation of a count-controlled loop very well.

The variable used to count must be initialized, it must be tested against some ending value, and it must be incremented each time through the loop. These three steps match the three parts of the `for` loop almost exactly.

In `RescueMission` in Chapter 8, the `getLowestSwimmer` method modified our standard count-controlled loop:

```

204  for (int row = 0; ((lowest == null) && (row != ROWS)); ++row) {
205      for (int column = 0; ((lowest == null) && (column != COLUMNS));
206          ++column) {

```

```

207     Swimmer curr = swimmers.get(row).get(column);
208     if (curr != null) {
209         lowest = curr;
210     }
211 }
212 }
213 return lowest;
214 }
215
216 /**

```

Listing 9.7: RescueMission: getLowestSwimmer

In lines 206 and 207 the Boolean expression for *<continuation>* extends the standard count-control test of having reached some bound with a test for `lowest` being `null`. The loop runs for some number of iterations or until the body of the loop sets `lowest` to a non-`null` value. A loop which runs over and over until some condition is met is known as a *sentinel-controlled* loop.

A sentinel is a guardian or, in our case, a marker. It marks the end of some phase of processing. Imagine writing a method to generate a random multiple of 3 between 0 and some given positive value, `bound`⁵.

If we ignore the need to have a multiple of 3, the method is fairly simple to write:

```

public int randomMultipleOfThree(int bound) {
    int multipleOfThree = Game.getCurrentGame().randomInt(bound);
    return multipleOfThree;
}

```

This is just a wrapper around a call to `randomInt` in the current game. This method could be defined in a game or a sprite, any class which imports `fang.core.Game`. The value returned may or may not be an actual multiple of 3. How can we make sure it is?

The `randomInt` method returns a number on the range `[0- bound)`. What are the odds that that number is a multiple of 3? If all numbers are equally likely, then one in three should be a multiple of 3. So one third of the time the above method should work. Is that good enough? No, we want it to work every time.

So, if we call `randomInt` 3 times, if each choice has a one in three chance of being a multiple of 3, we have a one in one chance of getting a multiple of 3, right? **Wrong**. With random numbers there is a chance that any number of them in a row is not a multiple of 3. The chance gets smaller and smaller the more times we select a random number but the number of times we must select *cannot be known* when the method begins. Thus this cannot be a count-controlled loop.

We will use a new construct, the `while` loop. First we will show an example loop inside `randomMultipleOfThree` and then we will present the template.

```

public int randomMultipleOfThree(int bound) {
    int multipleOfThree = Game.getCurrentGame().randomInt(bound);
    while (!(multipleOfThree % 3 == 0)) {
        multipleOfThree = Game.getCurrentGame().randomInt(bound);
    }
    return multipleOfThree;
}

```

A `while` loop takes a single Boolean expression after the word `while`. When the `while` line is executed, the expression is evaluated. If it evaluates to `true` then the body of the loop is executed; at the end of the body of the loop, execution returns to the `while` line, reevaluating the Boolean expression. When the Boolean expression evaluates to `false`, execution skips over the body of the loop and continues execution with the following statement.

⁵There are alternative approaches to this problem which do not use iteration at all. Exploring them is left as an exercise for the interested reader.

```
<whileStatement> ::= while (<continuation>) {
    <body>
}
```

The **while** statement looks a lot like an **if** statement and some students confuse the two. It is important to keep in mind that a **while** statement *cannot* have an **else** clause. Also, the body of an **if** statement is executed zero or once (depending on the Boolean expression); the body of a **while** statement is executed zero or *more* times with an unbounded value for *more*.

So, looking back at `randomMultipleOfThree`, if the first number chosen is a multiple of 3, then `multipleOfThree % 3` is 0. The Boolean expression `((multipleOfThree % 3) == 0)` is **true** so the logical inverse (or not) of that is **false**. The key is we wrote an expression that is **true** at the sentinel value and then applied a logical not to it. The loop will run while the sentinel check does not evaluate to **true**.

deMorgan's Laws

Note that the Boolean expression in our `multiple of 3` method can be simplified: if we push the logical not, `!`, into the expression it modifies we can rewrite the expression according to the following:

```
(!(multipleOfThree \% 3) == 0) = ((multipleOfThree \% 3) != 0)
```

This removes a level of parentheses. It also reverses the logic from

```
while (!P) {
    ...
}
```

to

```
while (Q) {
    ...
}
```

where `P` and `Q` are Boolean expressions and `!P = Q` or `!P` and `Q` have the same truth values for all possible values of variables they reference. Studies have shown that positive logic is easier for human programmers to understand and maintain.[?]

August De Morgan (1806-1871) was a logician who was influenced by George Boole's work and formally stated a couple of simple rules, rules for simplifying expressions with logical not in them. The following two lines use `==` to show that the two expressions are logically equivalent.

```
!(P && Q) == !P || !Q
!(P || Q) == !P && !Q
```

So, how can we use this? Consider a **while** loop to pick a random number which is either a multiple of 10 or a multiple of 11 or both. The Boolean expression can be built up:

```
((n \% 10) == 0) // multiple of 10
((n \% 11) == 0) // multiple of 11
((n \% 10) == 0) || ((n \% 11) == 0) // either 10 or 11
(!(n \% 10) == 0) || ((n \% 11) == 0) // not 10/11
```

This expression makes sense in that it was developed by expressing the sentinel condition, the condition where the loop should stop, and then that expression was logically inverted. This is the way a sentinel-controlled loop is typically built up.

The expression is not the simplest form to read for following programmers. It would make sense to push the `!` into the expression. That is done by applying the second of DeMorgan's Laws:

```
(!(n \% 10) == 0) || ((n \% 11) == 0)
!(n \% 10) == 0 && !(n \% 11) == 0
((n \% 10) != 0) && ((n \% 11) != 0)
```

The expression is first rewritten as an and of the inverse of the two subexpressions. Then the not is pushed into the subexpressions (it would work if they were `&&` or `||` expressions as well). We will not spend a lot of time working on simplifying logical expressions though this is a skill a programmer should develop.

9.4 String Manipulation

In Section 2.2, we saw that internally digital computers work only with discrete integers. This design influenced the tasks early computers were put to: space and atomic physics including the first atomic bomb, ballistics tables for artillery, and payroll and tax calculations.

The deepest insight presented in that section was that arbitrary *types* of values can be encoded for storage and manipulation in a digital computer. Computers are a *universal medium* in that any other medium can be encoded into a digital representation. This insight was first used when business computing became standard and continues as audio, video, and even immersive 3D environments are digitized and distributed using networks of digital computers.

This section will focus on how strings, sequences of letters, digits, and punctuation, or, more simply, sequences of characters, are encoded and stored inside the computer. We will begin looking at the `char` type (and the corresponding object type, `Character`). Then we'll look at the general public interface of the `String` type, and then look specifically at how strings are compared for ordering.

The `char` Data Type

Back in Chapter 2, the comparison was made between printable characters and *bytes* in the computer. A byte, eight binary digits or bits, can hold 256 different values. If you consider just the characters in a textbook (ignoring for the moment color or font values), how many different characters are you likely to see?

If the language of the text is English, the alphabet accounts for 52 letters (upper and lower case), common punctuation probably account for another 18 characters, and the digits account for another 10. That gives about 80 characters. Why encode characters in 256 values when 128 will do? Or, if we were less liberal with punctuation and mixed case, perhaps even as few as 64 different values.

There are two answers to the question of why not: memory is free; powers of two powers of two are powerful. Modern technology makes RAM memory and other memory devices very inexpensive. Thus saving 12.5% in memory costs more in convenience than it saves in memory. The other reason is that a byte is 8 bits and 8 is a power of 2. Thus $256 = 2^8 = (2^2)^3$. This has benefits in building and programming binary circuitry at the heart of a modern computer.

Given that we have more encodings than characters in English, we can extend the characters encoded to include those used in most Western European languages by adding accents, umlauts, and some special characters. The characters are encoded using the ASCII character coding sequence⁶.

The “Western European” limitation in the previous paragraph is indicative of where computers were developed and deployed. Other alphabets such as Cyrillic, Greek, Tamil, Thai, Arabic, Hebrew, and the like, are used in many modern programs by using a two byte character encoding scheme (known as Unicode). Operating with Unicode characters is an important part of internationalizing computer programs but, with that said, we are not going to discuss Unicode characters any further in this text. Unicode embeds the ASCII encoding to simplify working with Western European encoded legacy data.

How are characters encoded? That is, given a character, let's say 'A', what value is stored in a byte to represent it? The short answer: who cares?

We don't care what value 'A' has⁷ because that is an implementation detail. In fact, there are only three important pieces of information to remember about the ASCII encoding sequence:

1. The capital letters are contiguous. 'A' has an encoding one smaller than that of 'B' which is one smaller than that of 'C' and so on.
2. The lowercase letters are also contiguous.. 'a' is one smaller than 'b' which is one smaller than 'c'.

⁶American Standard Code for Information Interchange

⁷Though all of the authors do waste some portion of their brains remembering how 'A', 'a', and '0' are encoded.

Method Name	Description
<code>isDigit(char ch)</code>	true if <code>ch</code> is a digit; otherwise false .
<code>isLetter(char ch)</code>	true if <code>ch</code> is a letter; otherwise false .
<code>isLowerCase(char ch)</code>	true if <code>ch</code> is a lowercase letter; otherwise false .
<code>isUpperCase(char ch)</code>	true if <code>ch</code> is an uppercase letter; otherwise false .
<code>isSpaceChar(char ch)</code>	true if <code>ch</code> is the space character; otherwise false .
<code>isWhitespace(char ch)</code>	true if <code>ch</code> is considered whitespace by Java; otherwise false .
<code>char toLowerCase(char ch)</code>	returns lowercase equivalent to <code>ch</code> if it is a letter; otherwise returns <code>ch</code> unchanged.
<code>char toUpperCase(char ch)</code>	returns uppercase equivalent to <code>ch</code> if it is a letter; otherwise returns <code>ch</code> unchanged.

Table 9.1: Character Classification and Translation Routines

- The digit characters are contiguous. '0' is one smaller than '1', '0' is two smaller than '2', and nine smaller than '9'.

Remembering that a character is stored in a single byte (ignoring Unicode) and that there are three contiguous ranges in the encoding, ['A' - 'Z'], ['a' - 'z'], and ['0' - '9'] are the only implementation details we need to remember.

Character

The `Character` type is like `Integer`, `Double`, and `Boolean`, an object type which lets us create `ArrayLists` and provides many utility functions. Among the methods provided by `Character` are a series of static character classification routines. Some of these are summarized in the table below:

The value returned by `Game.getKeyPressed()` is a `char` so these methods can be applied to the values entered by the user. Imagine we wanted to call `setGameOver` with **true** if the user pressed either an uppercase or lowercase 'q' (for 'quit'). The following `if` statement makes that easy:

```
if (Character.toLowerCase(getKeyPressed()) == 'q') {
    setGameOver(true);
}
```

By forcing the case to be lower, we effectively ignore case. Notice that we use `==` to compare `char` values. It is also possible to use less than and greater than comparisons for individual characters. For example, we could write `myIsUpperCase(char ch)` as follows:

```
boolean myIsUpperCase(char ch) {
    return (('A' <= ch) && (ch <= 'Z'));
}
```

The `Boolean` expression is **true** when the value in `ch` (the encoding for the letter) is greater than or equal to the encoding of 'A' and when the value in `ch` is less than or equal to the encoding of 'Z'. Using the first rule of ASCII, that the uppercase letters are contiguous, any capital letter must be encoded between those values and only those 26 values encode uppercase English letters. This version of `isUpperCase` only handles English letters, not extended Western European letters (e.g., 'Ð' and 'Ê') nor Unicode characters; the `Character` version does handle those characters.

We will look at `char` in terms of sequences of them, the `String` type.

String: The Public Interface

A Java `String` is an *immutable* sequence of `char` values. *Immutable* simply means unchangeable; once a `String` has been constructed, it is not possible to change individual characters. Consider:

```
String one = "cat";
String two = "fish";

one = one + two;
```

How many String objects are created by this code? Is it even legal (if a String is immutable, how does the last line work?)?

This code is legal because both `one` and `two` are *references* to String objects. Thus the last assignment just changes *what* String `one` refers to without changing the String containing "cat". The first line constructs a literal String (all doubly quoted strings in Java code are converted to constructor calls by the compiler automatically). The next line constructs a second literal string. The last line calls the `+` operator for String. `+` takes two String operands and returns a new String containing the characters of the first operand followed by the characters in the second operand. `one` ends up referring to the third String constructed by this code, a String containing the characters "catfish".

A String is sequence of characters. This means we can write a loop to iterate over the characters in a string. First, we will look at a program that takes an unnamed command-line argument and prints out the characters in the argument, one character per line with single primes around the characters. We will do this in FANG.

```
1 import fang.core.Game;
2 import fang.util.InitializeApplication;
3
4 import java.util.ArrayList;
5
6 /**
7  * Read unnamed arguments, print out the first argument one character
8  * per line. Demonstrates how a String is a collection of char.
9  */
10 public class ArgumentCharacters
11     extends Game {
12     /**
13      * If there are any unnamed command-line parameters, then get the
14      * first one and print out the characters, one per line. If there are
15      * no parameters, print out a message to that effect.
16      */
17     @Override
18     public void setup() {
19         System.out.println(getClass().getName());
20         ArrayList<String> args = InitializeApplication.getUnnamedArgs();
21         if (args.isEmpty()) {
22             System.out.println(" No unnamed argument provided");
23         } else {
24             String firstArgument = args.get(0);
25             for (int i = 0; i != firstArgument.length(); ++i) {
26                 System.out.println(" str.charAt(" + i + ") = '" +
27                     firstArgument.charAt(i) + "'");
28             }
29         }
30     }
31 }
```

Listing 9.8: ArgumentCharacters

This program has a new feature at line 21: rather than typing in the name of the program's class (as we have done up until this point), the program takes advantage of Java's runtime class identification. Java knows what class an object is and keeps a reference to a `Class` object for that class; getting the class of the current object is as easy as calling `getClass`. The `Class` class is not always easy to follow but the `getName` method does just what you would think it would: it gets the name of the class, the *actual* class of the object on which `getClass` was called.

Thus the first line of output, the name of the program, is generated by printing the `String` returned from `getClass().getName()` in line 21.

Then, using the `FANG` method for getting command-line arguments, it checks to make sure there is at least one command-line argument. If there are none, an error message is printed and setup finishes. Otherwise, the first command-line argument is fetched and referred to with `firstArgument`. Each character in `firstArgument` is then retrieved using the `charAt` method with an index. Each character is printed on its own line using `System.out.println`. Sample output is given below.

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    ArgumentCharacters
ArgumentCharacters
No unnamed argument provided

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    ArgumentCharacters alpha bravo charlie
ArgumentCharacters
str.charAt(0) = 'a'
str.charAt(1) = 'l'
str.charAt(2) = 'p'
str.charAt(3) = 'h'
str.charAt(4) = 'a'

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    ArgumentCharacters "alpha bravo charlie"
ArgumentCharacters
str.charAt(0) = 'a'
str.charAt(1) = 'l'
str.charAt(2) = 'p'
str.charAt(3) = 'h'
str.charAt(4) = 'a'
str.charAt(5) = ' '
str.charAt(6) = 'b'
str.charAt(7) = 'r'
str.charAt(8) = 'a'
str.charAt(9) = 'v'
str.charAt(10) = 'o'
str.charAt(11) = ' '
str.charAt(12) = 'c'
str.charAt(13) = 'h'
str.charAt(14) = 'a'
str.charAt(15) = 'r'
str.charAt(16) = 'l'
str.charAt(17) = 'i'
str.charAt(18) = 'e'
```

What else can you do with a `String`? In Listing `lst:Rules:StringValueDemonstration`, `StringValueDemonstration.java`, we saw the `toUpperCase()`, `toLowerCase()`, and `substring(start, end)` methods. The first two methods return a copy of the `String` on which they are called, one containing the

`Character.toUpperCase(ch)` result for every character in the original string (or `Character.toLowerCase(ch)`, as appropriate).

`substring(start, end)` takes two `int` values and returns a new `String` containing the character sequence starting at the given `start` location (note in previous output shows `String` like `ArrayList` and arrays are indexed from 0) up to but not including the end position.

There is an overloaded definition of `substring(start)`; it returns a new `String` containing the character sequence starting at the `start` location and continuing to the end of the `String`. (Quick review: an *overloaded* method is one where there are more than one with the same name, only differentiated by their signatures. In this case, one version has two integer parameters, the other only one.)

The `indexOf` method takes a `String` parameter and returns the index of the left-most occurrence of the parameter in the `String` on which `indexOf` was called or `-1` if there is no match. There is an overloaded version that takes an character and returns the index of the left-most occurrence or `-1`. Each has another overloaded variation which takes a second parameter, an index where the search should start. It will return the left-most occurrence *at or after* the given index. `indexOf` is used in `GuessableWord` to count the number of underscores remaining in the word to guess.

```

121     int underscoreCount = 0;
122
123     /** where is the most recent underscore found */
124     int justMatchedIndex = showWord.indexOf('_', 0);
125     while (!(justMatchedIndex == -1)) { // negative logic is bad!
126         ++underscoreCount;
127         justMatchedIndex = showWord.indexOf('_', justMatchedIndex + 1);
128     }
129     return underscoreCount;
130 }
131 }

```

Listing 9.9: `GuessableWord`: `unguessedLetterCount`

On line 126, `justMatchedIndex` is initialized to the index of the left-most underscore in the field `showWord`; `showWord` is initialized to contain an underscore and a space for each letter to be guessed (double-spaces the word to make it easier to count the number of characters for the player). To count the number of letters as yet unguessed, this method loops through the shown word, matching `'_'` characters. Each time a match is found, a counter is incremented. When no more matches are to be found, the current count is the number of underscores and is returned.

Line 127 is a sentinel-controlled loop. For teaching purposes, to make it clear that it is a sentinel-controlled loop, the condition on the `while` loop is left in a negative logical form. The sentinel is a location of `-1` because `-1` is never a valid index into a `String`.

When designing a sentinel-controlled loop and choosing a sentinel, a good rule of thumb is to use a value which *cannot* appear as a legitimate value. If the sentinel *could* be a legal value, how can your program tell the difference between a legitimate and a sentinel occurrence? Once you define the context necessary to differentiate between the two values you have defined your actual sentinel value, the contextualized occurrence of the value which is never a legitimate value.

`String` is special in Java in that it is the only object type for which you can directly write literal values. One reason that it is immutable is so that it behaves as much as possible like a plain-old data type. The biggest departure in behavior between POD and `String` is in comparing values.

`String.compareTo(String)`

Consider writing a method which takes two `int` parameters and returns the larger of the two values. Consider how you would write this method, `myMax` for a moment before you read on.

The method should return `int` as well as take two `int` parameters. Since the parameters are arbitrary values (they are not heights or coordinates or anything), `a` and `b` make sense as names. The body of the method should be an `if` statement comparing the two numbers and returning the larger value:

```

int myMax(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

```

Convince yourself that this works by walking through the code with a few different values. In particular, what happens when the two integers are equal?

Looking at the code, we see a Boolean expression using the `>` comparison operator. If we wanted to overload `myMax` so that it worked with `char` or `double`, the plain-old data types, the body of the overloaded methods would look identical; only the signature line would change.

Since `String` is supposed to behave like a POD, it is tempting to try comparing two `String` using `>` (or `==` or even `==`). Java will not compile the following code:

```

String myMax(String a, String b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

```

The compiler gives the following error message (this output was edited to put the error message in the middle of the code):

```

String myMax(String a, String b) {
    if (a > b) {
        ^
        StringCompare.java:5: operator > cannot be
        applied to java.lang.String, java.lang.String
        return a;
    } else {
        return b;
    }
}

```

The comparison would *compile* with `==` or `!=` in the middle but it would not mean what you probably think that it would mean (more on that below).

Java defines a special **interface** called `Comparable` and all `Comparable` objects (`String` is one) provide a `compareTo` method. The method takes a `String` as its parameter and compares the string on which `compareTo` is called to the parameter, returning a negative number, 0, or a positive number, depending on the order of the two strings: if called on string is before parameter, return a negative number; if called string and parameter are equal, return 0, and if called string should come after the parameter, return a positive number. Thus the following rewrite of `myMax` works for `String`:

```

String myMax(String a, String b) {
    if (a.compareTo(b) > 0) { // a after b
        return a;
    } else {
        return b;
    }
}

```

There is also an `equals` method which returns `true` if two `String` objects contain the same sequence of characters and `false` otherwise. Thus the following expressions are all `true`:

```
"stop".equals("s"+"top")
("abc" + "def").equals("abcdef")
"catfish".substring(3).equals("fish")
```

It is important to note that the following expression returns `false` on the author's machine:

```
"catfish".substring(3) == "fish"
```

This is because `==`, when applied to object or *reference* types, compares the references, not the referents. So, since `"fish"` is a `String` constructed by a call to `new` before the program starts running and the result of the call to `substring` is a newly created `String` copying the letters 'f', 'i', 's', and 'h' out of a different statically constructed literal `String`, they are not the exact same object so the `==` comparison returns `false`.

While `==` and `!=` can be used to compare objects, the general rule is that they should *not* be used to compare objects. All objects have an `equals` method and most types implement a meaningful comparison when they override the version provided by `Object`.

What Order?

What order is used when comparing `String` with `compareTo`? The `myMax` method neatly sidestepped needing to know. In Java, `Strings` are compared in *lexicographic order* which means they are compared in dictionary order. If two strings contain only lowercase or only uppercase letters, then they are sorted into alphabetic order according to the left-most character which differs between them. Thus `"cab"` comes before `"cat"` and `"capacitor"` because 'b' comes before 't' or 'p'. If one string is a prefix of the other (that is, the letters in one of the strings runs out before a differing character is found), then the shorter word comes first. Thus `"cab"` also comes before `"cabal"` and `"cable"`, too.

The "only lowercase or only uppercase" stipulation above was to make the examples easy to include. What actually happens is the two strings are compared, character by character, from left to right. As soon as a position differs, the two `Strings` compare in the order of the differing characters. That is, for ASCII characters, whichever of the characters has a smaller ASCII code comes first. Because uppercase letters are contiguous and lowercase letters are contiguous, this works out to being alphabetic order as long as we don't compare uppercase to lowercase letters⁸.

The Java `String` class also provides the `compareToIgnoreCase` method which ignored differences between uppercase and lowercase letters.

It is worthwhile to look at the documentation for the `java.lang.String` class because it is a very central class, one used for a lot of useful things. We will now look at how we can read a text file on a computer into a `String` or a list of `String`; once we can do that, having Hangman pick a different word is as easy as reading a list of words and picking one at random.

9.5 Reading Files

Section 2.2 introduced random access memory (RAM), the memory inside of the computer, and disk drives, non-volatile memory stored outside of the RAM memory. Because the disk drive's storage capacity is much larger than the RAM and the user stores different things on the drive, things they want to be able to find again, to send to others, and to reopen in various computer programs, the disk drive has a *file system*, a method to add user specified names to locations on the disk drive.

This section covers how to interact with the file system, opening a specific file for input and reading the values from a file into a running Java program.

⁸For those interested, all uppercase letters sort earlier than any lowercase letter. Digits sort before any uppercase letter.

Opening a File for Input

There are two steps in reading input from a text file: connecting a Java data structure to a specific file on the disk (or on the network) and reading and disassembling the contents of the file. We will be introducing two classes in this section, one specifically for each of the phases: `java.io.File` to provide an internal representation for a file system object and `java.util.Scanner` for scanning through the contents of the file, extracting each word or each line.

Along the way we will see a new Java construct, the `try...catch` block. This is a construct that is a lot like an `if/else` statement except that the first block, the `try` part is always run and the `catch` part is executed *only* if there is an exception while running the `try` block. An exception is an error or some other unexpected condition.

Specifying a Filename

Think, for a minute, about your hard drive (or, if you prefer, your USB thumb drive or a networked drive to which your computer is connected). It is composed of a large number of *sectors*, units which are addressed by numbers (the exact nature of the numbering scheme is unimportant).

The computer operating system permits allocation of the sectors for use in *files* and *folders*; a folder can also be called a *directory*. Modern graphical user interfaces use a *desktop metaphor* where files are stored together in folders which can, themselves, be arranged in other folders in a hierarchical storage system. Thus you might keep your Java programs together in a `CS101` folder which is inside of a `SchoolWork` folder.

When specifying a file name on the command-line, you will use slashes between folder and file names. Thus, from the point of view of the `SchoolWork` folder the `NewtonsApple` source file would have the name `CS101/NewtonsApple.java`⁹. The concatenation of folders and finally a file name is called a *file path* or *path name*.

A path name can begin with a slash as in `/home/blad/SchoolWork` in which case the hierarchy is anchored at the *root* of the file system¹⁰. A path name starting with a slash is an *absolute path name*; no matter where you are inside the file system, an absolute path name will always refer to exactly the same file on the system.

If a path name does *not* begin with a slash, the name is a *relative path name*; the hierarchy starts at the *current directory* and searches downward from there. The first example here specified that the path `CS101/NewtonsApple.java` applies if you start in the `SchoolWork` folder.

We will be specifying file names using relative path names. When you run a Java program, the current folder is the folder where you run it. Thus if I run `ExistsFile.java` with the following command-line:

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar ExistsFile pets.txt
```

and the command-line parameters are treated as file names, the file specified is `~/Chapter09/pets.txt`.

What is `ExistsFile`? It is a program which demonstrates how to **import** the `File` class in Java, how to call the constructor, and how to query a `File` object to see if it refers to a file which exists on the system. The code is as follows:

```
1 import java.io.File;
2
3 /**
4  * A program which reads its command-line arguments, treating each as
5  * the name of a file. Create an ExistsFile object with the name of the
6  * file. Then check if the file exists.
7  */
8 public class ExistsFile {
9     /** Java reference attached to file on file system */
10    File file;
```

⁹Microsoft Windows derivatives have a file system ported from Unix with some changes. The most noticeable difference is the use of `\` instead of `/` between levels in the file name hierarchy. This book will consistently use the `/`; feel free to translate it as you read.

¹⁰On Microsoft systems this is also done with a letter followed by a colon as in `C:`. This is known as a disk name. Microsoft operating systems have multiple roots to their file system.

```

11
12  /**
13   * Create a new ExistsFile with the file pointed to the named file.
14   *
15   * @param fname the name of the file to connect to; the named file
16   *             need not exist (it will be checked in exists()).
17   */
18  public ExistsFile(String fname) {
19      file = new File(fname);
20  }
21
22  /**
23   * Check if the file exists, printing an appropriate message.
24   */
25  public void exists() {
26      if (file.exists()) {
27          System.out.println "\"" + file.getName() + "\" is a file.");
28      } else {
29          System.out.println "\"" + file.getName() + "\" is NOT a file.");
30      }
31  }
32
33  /**
34   * The arguments are assumed to be the names of files. Each is opened
35   * and echoed to the screen.
36   *
37   * @param args command-line arguments; should name files to be
38   *             checked for existence
39   */
40  public static void main(String[] args) {
41      for (int argNdx = 0; argNdx != args.length; ++argNdx) {
42          ExistsFile nextFile = new ExistsFile(args[argNdx]);
43          nextFile.exists();
44      }
45  }
46 }

```

Listing 9.10: ExistsFile

The main method is modeled on the one used to cycle across command-line parameters. For each command-line parameter, an `ExistsFile` object is constructed. On line 21 that constructor calls the `File` constructor. The most common version of the `File` constructor takes a `String` containing a path name. The constructed `File` object then refers to that location in the file system.

In the program, line 45 calls the `exists` method on the newly created `ExistsFile` object. All that method does (see lines 28-32) is call the `exists` method of the `File` object. That Boolean method returns `true` if the file exists and `false` if it does not.

Thus the output of the previously shown command-line is:

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar ExistsFile pets.txt
"pets.txt" is a file.
```

This assumes that there is a file with the given name in the current folder.

try... catch and Exceptions

The file `EchoFile.java` is almost identical to `ExistsFile.java`. The difference is that the `exists` method is replaced by an `echo` method:

```

23     file = new File(fname);
24 }
25
26 /**
27  * Echo the file to standard output. If there is a problem with the
28  * file not existing or not being readable by the current user, an
29  * appropriate error message should be printed.
30  */
31 public void echo() {
32     if (file.exists() && file.canRead()) {
33         try {
34             Scanner echoScanner = new Scanner(file);
35             String line;
36             while (echoScanner.hasNextLine()) {
37                 line = echoScanner.nextLine();
38                 System.out.println(line);
39             }
40             echoScanner.close();
41         } catch (FileNotFoundException e) {

```

Listing 9.11: EchoFile: echo

If the file exists (as in `ExistsFile`, `file` is a field which was initialized in the constructor) and the file is readable, then we go into the body of the `if` statement. If either of the conditions is not met, then we print an error message and return from the method.

We will now look at lines 25-36 at two levels of abstraction. First we will look at how a file is opened and read line by line. Assuming all goes well, lines 26-32 run and the contents of the file are copied onto the screen. Line 26 creates a `Scanner` associated with `file`. A `Scanner` is an internal Java representation of a text file. The `Scanner` keeps track of a *current position* inside the file it is reading. As values are read from the `Scanner`, the current position moves through the input file; a `Scanner` can read the next word or line *and* can be queried to see if there is anything more to read at the current position.

Thus the `while` loop on lines 28-31 is a sentinel-controlled loop which runs until there are no more lines of input to be read. Each time through the body of the loop, `line` is set to the contents of the next line in the file the `Scanner` is scanning and then `line` is printed to standard output.

This loop is a special sentinel-controlled loop, an *end-of-file-controlled (eof-controlled) loop*. The `hasNext` and `hasNextLine` methods return `true` so long as the current position in the file is not at the end of the file. Thus the loop ends when the input file is exhausted.

Assuming `pets.txt` contains the names of one author's family pets, the output of running this program would be:

```

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar EchoFile pets.txt
Explorer
Frodo
Grace Murray Hopper

```

What could go wrong? When associating a `Scanner` with a file in the file system many things could go wrong: the file might not exist (because some folder in the path name does not exist or the file itself does not exist); the file might be protected so only authorized users can read it (imagine a password file); even if the file exists and is readable, something could change while the program is running which changes things (a different program could delete or replace the file).

Imagine that every call to `next` had to be followed with a check to see if there were any errors that the `Scanner` knew about. Then, after you see that that just about doubles the size of this loop, imagine a similar `if` statement for the constructor call and another to make sure `close` worked.

There would be a lot of `if` statements. So many, in fact, that a lot of programmers would skip putting them in: most of the time, if the file exists, reading the whole thing finishes with no problems. So if we just check on the constructor, then we are “safe”. Of course we aren’t really safe; we just get away with ignoring errors most of the time. When errors happen, our program crashes with ugly error messages.

Many older programming languages used a system similar to this and the errors were routinely ignored. To avoid this *and* to keep the code easy to read, Java implements *exceptions*. An exception is a special signal which Java can **throw**. In reality it is just another object type. When a method can throw an exception the Java compiler requires the programmer to wrap the call to that method in a **try...catch** block. The **try** part is executed. If any problem occurs, an exception is thrown and the **catch** block runs with the parameter set to the exception thrown.

Thus if the file named does not exist, the Scanner constructor will throw an exception, one specifying that the file does not exist. We specifically catch that kind of exception on line 33 and the block on lines 33-36 will execute with `e` set to the exception thrown.

The `printStackTrace` method of exceptions will print out where the error occurred (by giving what Java methods were executing when it occurred). Robustly dealing with exceptions is very sophisticated; this book will catch them and, in general, end the current task or, if nothing more can be done, end the program.

Reading a File into a Collection

How can you read the contents of a text file into a collection? We will write two variations on this theme: one that reads the file word by word and one that reads the file line by line. Each program will get the name of the file (only one) to read from the command-line. We will then read the file using an appropriate eof-controlled loop, adding elements to a list. Finally we will print out all of the entries in the list so we can see how the file was read. Then this section will end and we will finish Hangman so that it gets a list of phrases from a text file.

Reading a Text File by Word

The main method is a lot like that in the echo and exists programs. The biggest differences are that the program expects exactly one file name command-line argument. Thus we check the size of the argument array and only do the work if there is a single argument. If the number of arguments is wrong, print an error message and let main finish.

```

80     String fname = args[0];
81     TextFileByWord byWord = new TextFileByWord();
82     byWord.listLoadFromFile(fname);
83     System.out.println("----- listPrint -----");
84     byWord.listPrint();
85     System.out.println("----- listPrint -----");
86     System.out.println("There were " + byWord.listSize() +
87         " words in " + fname);
88 } else {
89     System.out.println(
90         "Usage: provide exactly one (1) file name on command-line.");
91     System.out.println("Program Terminating");
92 }
93 }
94 }
```

Listing 9.12: TextFileByWord: main

The other difference is that the `TextFileByWord` object has a larger public interface: the default constructor, the `listLoadFromFile` method (which now takes the file name), a `listPrint` method, and a `listSize` method. This is an attempt to make sure each method has a single, well-defined purpose. While the printing could have been included directly in the `loadListFromFile` method, it would have made using the same code

in a program that did not print out the contents impossible (or require a new and different load method). Single purpose methods mean we can combine them as we see fit in the order we choose to call them.

`listSize` and `listPrint` are fairly obvious, being based on code we saw when we started working with single dimensional `ArrayLists`:

```

17  /**
57      System.out.println(strings.get(i));
58  }
59  }
60
61  /**
68  }
69
70  /**

```

Listing 9.13: `TextFileByWord`: the collection

Given that `strings` refers to an `ArrayList<String>` (which it does because one is assigned to it in the constructor (not shown) and the value is reassigned with a file is read (just below)), the `listSize` method just returns the size of the list and the `listPrint` method prints out the contents of the list. Again, it would have been possible to have the marker lines (lines 88 and 90 in `main`) as part of `listPrint`; why is that a bad idea?

Finally we get to the meat of the program, `listLoadFromFile`. This code is based on that seen in `echoFile` earlier in this chapter.

```

32  if (file.exists() && file.canRead()) {
33      try {
34          ArrayList<String> localStrings = new ArrayList<String>();
35          Scanner scanner = new Scanner(file);
36          String word;
37          while (scanner.hasNext()) {
38              word = scanner.next();
39              localStrings.add(word);
40          }
41          strings = localStrings;
42          scanner.close();
43      } catch (FileNotFoundException e) {
44          System.out.println("PANIC: This should never happen!");
45          e.printStackTrace();
46      }
47  } else {
48      System.out.println("Unable to open \"" + fname + "\" for input");
49  }
50  }
51
52  /**

```

Listing 9.14: `TextFileByWord`: `listLoadFromFile`

The method creates a `File` and then creates a `Scanner` associated with that file (if the file exists and can be read). The `try...catch` construct is just as we have used it before. Assuming the loop finishes (without an error), the scanner is closed as all files should be closed and, because there was no error, the `strings` field is set to refer to the newly filled in list.

Notice that to read *word-by-word*, the `Scanner` methods `hasNext` and `next` are used to check for and read the next *token*. By default, `Scanner` breaks its input up on whitespace; thus each value returned by `next` is a string of non-whitespace characters. The output of this program, when run on `pets.txt` is:

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    TextFileByWord pets.txt
----- listPrint -----
Explorer
Frodo
Grace
Murray
Hopper
----- listPrint -----
There were 5 words in pets.txt
```

Why five when there are only three pets? Because words are separated by whitespace, the last pet name, “Grace Murray Hopper” is parsed as three separate tokens. Looking further up at the echoed content of the file, there are five separate words, three on the third line. Thus this output matches what we would expect.

What if we used a different file, say numbers.txt?

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    EchoFile numbers.txt
100 101
3. 1415
1000000

eins 2 THREE IV
```

Key things to note about the contents of the file: there is a mix of “numbers” and “words”. But the Scanner uses whitespace to mark out words. Thus there are only words, some of which have digits as characters. How many words are there in numbers.txt?

```
~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    TextFileByWord numbers.txt
----- listPrint -----
100
101
3. 1415
1000000
eins
2
THREE
IV
----- listPrint -----
There were 8 words in numbers.txt
```

It is worth noting that Scanner does have `nextInt` and `nextDouble`, both capable of reading text representation of a number and converting it to an `int` or `double` with the appropriate value; we will work with them in the following chapters.

Reading a Text File by Line

To read a file line-by-line with a Scanner is almost the same as reading it word-by-word. The `next` and `hasNext` methods are replaced with calls to the `nextLine` and `hasNextLine` methods. `TextFileByLine.java` is the same as `TextFileByWord.java` except for the contents of the `listLoadFromFile` method:

```
32     if (file.exists() && file.canRead()) {
33         try {
34             ArrayList<String> localStrings = new ArrayList<String>();
35             Scanner scanner = new Scanner(file);
```

```

36     String line;
37     while (scanner.hasNextLine()) {
38         line = scanner.nextLine();
39         localStrings.add(line);
40     }
41     strings = localStrings;
42     scanner.close();
43 } catch (FileNotFoundException e) {
44     System.out.println("PANIC: This should never happen!");
45     e.printStackTrace();
46 }
47 } else {
48     System.out.println("Unable to open \"" + fname + "\" for input");
49 }
50 }
51
52 /**

```

Listing 9.15: TextFileByLine: listLoadFromFile

Running this program generates the following output:

```

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    TextFileByWord pets.txt
----- listPrint -----
Explorer
Frodo
Grace Murray Hopper
----- listPrint -----
There were 3 lines in pets.txt

~/Chapter09% java -classpath ./usr/lib/jvm/fang.jar \
    TextFileByWord numbers.txt
----- listPrint -----
100 101
3.1415
1000000

eins 2 THREE IV
----- listPrint -----
There were 5 lines in numbers.txt

```

The key things to note are that “Grace Murray Hopper” is treated as a single line (the line was read *including* the embedded whitespace) and the fourth line in `numbers.txt`, a blank line, is read and treated as a line. Calling `nextLine` reads from the current file position to the end of the current line. The end of a line is marked by a special sequence of characters.

Among the characters in the ASCII sequence are non-printing *control* characters. One is the null character, ‘0’. Two others are the *carriage return* character, ‘r’, and the *new line* character, ‘n’. Different operating systems use different combinations of carriage return and new line characters to mark the end of the line¹¹. The Scanner uses the native version of end-of-line marker for the system on which it is running.

The combination of `try...catch` and the eof-controlled `while` loop using a Scanner is a very important Java idiom. Many program assignments in the remainder of this book call for reading a file of a given format. It is worth practicing with this construct until it becomes comfortable.

¹¹Unix (and Linux) uses r, MacOS uses n, and Microsoft operating systems use rn.

9.6 Finishing Hangman

Now it is time to finish the Hangman program. First we will have the program read a named file for potential words and select one from the list at random. We'll also remove each word as we use it so that we can avoid repeats. Then we will look at the `AlphabetSelector` object, in particular the code which checks for mouse clicks on sprites which are inside of a `CompositeSprite`.

Replacing the Stub

We now know how to get the command-line arguments provided to our FANG program, how to treat such an argument as the name of a file, and how to open that file for input, reading the contents into a collection.

Before we look at the code, how are we going to store the “words” in the data file? That is, are we going to play Hangman solely with single words or will we permit multi-word phrases? Looking at the popularity of game shows based on the same premise as Hangman, permitting either words or phrases makes sense.

If we will permit embedded spaces in a phrase, how does the program know where a phrase begins or ends? One phrase per line is easy to take apart with a `Scanner`. Thus we will define our data file as a text file containing one word or phrase per line.

Also, if we cannot read the file for input or there is no named file then we will end the program. This is because we will have any way to select a word for the player to guess.

First, the replacement `wordLoad` method. Note that the line numbers have moved up by two; this listing includes the declaration of `strings`, the list holding the words remaining to be chosen.

```

22
23  /** the alphabet selector on the screen */
154     listLoadFromFile(getFname());
155     }
156     int wordNdx = randomInt(0, strings.size());
157     return strings.remove(wordNdx);
158     }
159
160  /**

```

Listing 9.16: Hangman: `wordLoad`

If the list of strings is `null` (it has never been initialized) or it is empty, then we need to load the list from a file. The `listLoadFromFile` method is similar to that in `TextFileByLine`. It fills `strings`. When `strings` is not empty, pick a random index into it and remove the given entry. `remove` returns the value that is being removed so it is exactly what we want to return from this method.

```

168     if (args.size() > 0) {
169         return args.get(0);
170     } else {
171         return null;
172     }
173 }
174
175 /**
185     if (fname != null) {
186         File file = new File(fname);
187         if (file.exists() && file.canRead()) {
188             try {
189                 ArrayList<String> localStrings = new ArrayList<String>();
190                 Scanner scanner = new Scanner(file);
191                 String line;
192                 while (scanner.hasNextLine()) {

```

```

193         line = scanner.nextLine();
194         localStrings.add(line.toLowerCase());
195     }
196     strings = localStrings;
197     scanner.close();
198 } catch (FileNotFoundException e) {
199     System.out.println("PANIC: This should never happen!");
200     e.printStackTrace();
201 }
202 }
203 }
204 if (strings == null) {
205     System.out.println("There was an error reading the word file.");
206     System.out.println(
207         " Make sure to include the path to a word file.");
208     System.out.println(" Make sure file exists and is readable.");
209     System.exit(1);
210 }
211 }
212 }

```

Listing 9.17: Hangman: getFilename and listLoadFromFile

The `getFilename` method checks for unnamed command-line arguments. If there are any, the first is treated as the name of the word file and is the value returned by the method. If there are no unnamed command-line arguments, then the method returns `null`. This is noted in the header comment; because this value is passed directly to `listLoadFromFile`, that method must be able to handle a `null` file name.

The `listLoadFromFile` method checks whether the parameter is non-`null`. If it is, then the list loading code matches that seen `TextFileByLine` except for line 196: each phrase in the word list is forced to lower case. This is because the `AlphabetSelector` has no shift key.

Line 186 sets `strings` to `null`. If all goes well, it will no longer be `null` at line 206. If something goes wrong, it *will* be `null`. If it is `null` at line 206, the game cannot be played. A message is printed to standard output and the program terminates. `System.exit` is a method that ends a running application. The number passed to it is returned to the operating system as the program's exit code. By convention, a 0 return code means there were no problems and non-zero return codes mean a non-standard exit. Thus code 1 means there was an error.

The only remaining tricky code is in the `AlphabetSelector`.

Tracking Used Letters

In `RescueMission`, we used nested count-controlled loops to check each swimmer for intersection with the rescue ring. Almost the same code could have been used to determine whether the user had clicked the mouse inside of any swimmers. Conceptually, that is the basis of the `selectedChar` method in `AlphabetSelector`.

```

99     if (insideClick != null) {
100         for (int row = 0; row != letters.size(); ++row) {
101             ArrayList<LetterSprite> currRow = letters.get(row);
102             for (int column = 0; column != currRow.size(); ++column) {
103                 LetterSprite curr = currRow.get(column);
104                 if (curr.intersects(insideClick) && curr.isReady()) {
105                     selected = curr;
106                 }
107             }
108         }

```

```

109     }
110     return selected;
111 }
112
113 /**

```

Listing 9.18: AlphabetSelector: selectedChar

selectedChar returns `null` if the click passed in to the method is `null` or if the click does not intersect any letter sprite. If the click is non-`null` then each sprite in the two-dimensional list is checked to see if it intersects the mouse click *and* the sprite is in the ready state. A ready, intersected sprite is assigned to `selected`, the value returned at the end of the method; if it were possible for more than one ready sprite to intersect the same location, then the last one (in terms of indexes into the two lists) would be the value returned.

selectedChar is overloaded. The signature of the one shown here takes a `Location2D`, the FANG class representing two-dimensional points (such as where a mouse has been clicked). The other overloaded version has an empty parameter list.

```

75     Location2D outsideClick = Game.getCurrentGame().getClick2D();
76     if (outsideClick != null) {
77         Location2D insideClick = getFrameLocation(outsideClick);
78         LetterSprite letterSprite = selectedChar(insideClick);
79         if (letterSprite != null) {
80             letterSprite.startUsed();
81             letter = letterSprite.getLetter();
82         }
83     }
84     return letter;
85 }
86
87 /**

```

Listing 9.19: AlphabetSelector: selectedChar

Line 77 calls `getClick2D` on the current game. If the value is non-`null`, then it is translated from the game's coordinates to the `CompositeSprite`'s coordinates. Line 79 uses the `CompositeSprite` method `getFrameLocation`. The method takes a `Location2D` relative to the origin in the game with (0.0, 0.0) in the upper-left corner and transforms it into the coordinate system of the sprite where (0.0, 0.0) is located at the center of the sprite.

The transformation also takes into account rotation and scaling. This is why the parameter passed to `selectedChar (Location2D)` is called `insideClick`, to make sure to other programmers that the location must be in sprite coordinates.

If the value returned from `selectedChar (Location2D)` is non-`null` then the `LetterSprite` is set to the used state and the letter represented on the letter sprite is returned as the value of `selectedChar ()`.

9.7 Summary

Java Templates

Chapter Review Exercises

Review Exercise 9.1 TK

Programming Problems

Programming Problem 9.1

Console I/O: Games without FANG

We know half of what we need to know about standard console input and output. To date we have used `System.out` to print values to standard output. This chapter addresses the missing half by introducing how we read information from standard input. With both standard input and standard output, we can write a game which does not use FANG at all.

10.1 Another Dice Game: Pig

In this chapter we will implement a computer-mediated version of the folk dice game *Pig*. *Pig* is a jeopardy dice game, a term coined by Reiner Knizia in *Dice Games Properly Explained*[Kni00]. In a jeopardy dice game, player's decisions are between protecting current gains or risking current gains in search of greater gains.

In *Pig*, players take turns rolling a single 6-sided die. If they roll a 1 (a “pig”) they lose their turn and add nothing to their score; any other roll is added to their *turn total*. After any roll the player may choose to *hold* in which case their turn total is added to their score. The first player with a score greater than 100 wins.

A console game is limited to text input and text output (we will not be using the mouse, sprites, or buttons). Thus the design of *Pig* looks like this:

The design shows two columns on the screen; the actual program only uses `System.out` without any screen control (no way to reposition the output point to a given row/column) so the output will really be in one column scrolling off the top as more is printed below.

Notice that the design differentiates between things typed by the players and things printed on the screen by the program. Further, notice that whenever the program expects input from the user provides a *prompt*. As with any game we have designed it is important to keep the player informed as to what is expected of them (and when).

There are two phases of the game in the design. Initially the game asks for the players' names. When all the players are represented, some player types “done” and the initialization of the game is finished. After that the game is played by taking turns.

Looking at the design one can see the video game loop in action:

- Show the current player their roll
- Prompt the user for what they want to do
- Update the state of the game by either rolling again or holding the total for the turn

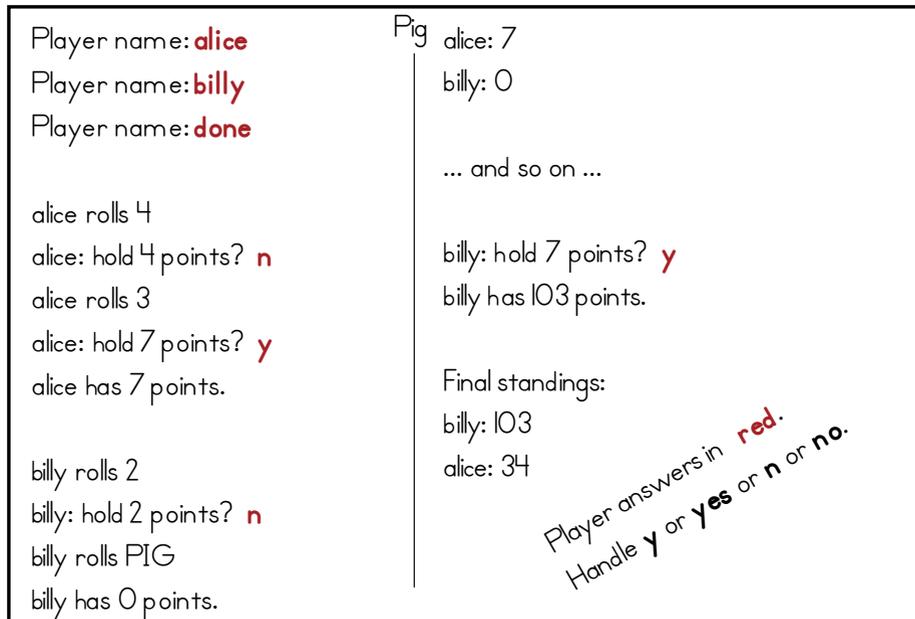


Figure 10.1: Design of Pig Game

Which player has control alternates by turn. When a player holds or rolls a pig the state of the game is updated and the next player's turn begins. Notice that each time around all of the players the game also displays the current standings (in the order in which players take their turns). When a player holds and puts their score over 100, the game is over and the standings are printed one last time, this time *sorted* in descending order by score.

The remainder of this section focuses on the design of the classes of `Pig`. Then the chapter introduces pure console I/O and the idea of sorting a collection.

Pig and its Classes

As we have seen before, a non-FANG program requires a `public static void main` method. In FANG the default version of this method constructs an object of the right class (the class which is specified on the command-line) and calls a particular method on it (`runAsApplication` though we will not use that name). Modeling our class on that, we have, as a first pass for a design, something like this:

```
public class Pig {
    public static void main(String args[])...
    private Pig()...
    private void play()...
}
```

Information Hiding

Why is the `Pig` constructor `private`? Because it *can* be. Another principle of software engineering is the Need to Know principle: limit the scope and visibility of all fields and methods as much as possible. If the program will work with a method declared to be `private`, declare the method to be `private`; if it works with `protected`, use `protected` rather than `public`. The point is to prefer the most restrictive visibility possible.

The `Pig` constructor can be `private` because it is only called in `Pig.main` (the `main` method declared inside the `Pig` class). That is, the only call to the constructor is in a method declared in the same class. Thus a `private` constructor is possible.

Limited visibility is desirable for two reasons, both related to design and, more importantly, redesign of the class. Changing the signature of a **private** method or the type of a **private** field can only impact the single Java file in which the change is made. That is, no other file need be edited or even recompiled. Isolating change like this makes changes much less complex.

Similarly, any special requirements of our class, say that the list of players never be **null** or some such, is documented in the Java file implementing the class *and* **private** fields and methods can only be modified by changing the source code containing them. This means the programmer should be working at the level of abstraction represented by the class when working on the **private** parts. All non-**static** methods of `Pig` are declared to be **private** for this reason.

What does `play` do? Looking at the design, there are three phases of the game: getting the players' names, playing the game, and finishing the game. The first and last phases are not game loops. Getting the players' names could be wrapped up in a method of its own as could announcing the winner of the game (that is what happens when the game ends). Playing the game is, itself, a video game loop. The extended design is:

```
public class Pig {
    public static void main(String args[])...
    private Pig()...

    // ----- main game phases -----
    private void getPlayersNames()...
    private void play()...
    private void announceWinner(Player winner)...

    // ----- video game loop -----
    private void showState(Player curr)...
    private void handleUserTurn(Player curr)...
    private boolean continueGame(Player curr)...

    // ----- player list handling -----
    private Player getPlayer(int n)...
    private int indexOf(Player p)...
}
```

Five methods take a `Player` object reference as a parameter and a sixth returns a `Player` object reference. What is a `Player`? We will design the class below. For the moment a `Player` holds all the information necessary to identify a player and know how they are doing in the game.

The three video game loop methods take the current player so that they can tell whose turn it is. `showState` shows the player's current score and, if this is the first player, shows the standings of all players. It is possible to show the standings before each player's turn but it seems to make more sense to show them once per round.

The `handleUserTurn` method has a player take a turn. Consider that taking a turn involves showing the player the state of their turn, letting them decide if they wish to continue, and then updating the state of their turn. Thus this "get user input" portion of the video game loop of `Pig` invokes another video game loop.

The `continueGame` method updates game state and returns **true** if the game should continue and **false** if the game is over. `Pig` ends when one player wins the game. Only one player's score can change during any given turn (the current player), so only one player needs to be checked if they won. In some games you must traverse the list of players in order to check if anyone has won.

The `getPlayer` and `indexOf` methods are for working with the list of `Player` objects. `getPlayer` takes an index into the list of `Players` and returns the `Player` object associated with that location. The `indexOf` method is the inverse function taking a `Player` in the list of `Players` and returning the index where that `Player` is. The name `indexOf` is a Java standard name for methods which convert from an element to its index (we have seen `String.indexOf` for finding the index of a `String` or `char` in a `String`, for example).

Now, what is a `Player`? Imagine that you were keeping track of an ongoing game of `Pig` with pencil and paper. What information would you keep track of? You need to know whose turn it is (that is the turn number passed to the various methods in `Pig`), the name and the score for each person playing.

When you find the need to group information together into a unit, that is when you should think of creating a class. It would be possible to design `Pig` without having a `Player` class but keeping the score and the name together in a single object makes life much simpler. What does the interface for the `Player` class look like?

```
public class Player {
    public Player(String name)...
    public String getName()...
    public int getScore()...
    public void takeTurn()...
    public String toString()...
}
```

When announcing the winning player, `Pig` needs access to the name and the score of the winning player. That is why the two `get*` methods are provided. `takeTurn` is the method which gets input from the user. As the name suggests, it actually does much more: it rolls the die, tracking the player's turn score, and, when the player chooses to hold, updates the player's score. It also handles the player rolling a pig and ending their turn without adding to their score. It is, internally, another, smaller video game loop.

The `toString` method is a method declared in `Object`, the class which *all* Java classes extend. If a class lists no parent classes, then they implicitly extend `Object`. The `toString` method returns a `String` representation of the object on which it is called. This permits `print/println` methods to print out any object. When a reference to an object is passed to `print` (as in `System.out.print`), the object's `toString` method is called and the result is printed to standard output.

This is an example of *polymorphism*, having many forms. The method has the signature `public void print(Object obj)` and internally it calls `obj.toString()`. Java calls the lowest overriding implementation of a method so if `obj` is *really* a `Player`, then the overridden `toString` is called.

Formal, Actual, Static, Dynamic

In the *signature* and definition of `public void print(Object obj)`, the parameter `obj` is a *formal* parameter. It is said to have a *static* type of `Object`¹; a static attribute is any attribute of the program which could be known at *compile time*.

When `print` is called, as in `System.out.println(somePlayer)`, the parameter `somePlayer` is an *actual* parameter. In the scope where the call is made, `somePlayer` has a static type which can be determined from the declaration: `Player somePlayer`; The static type is `Player`. Because `somePlayer` is passed into the parameter `obj`, both of these references share *dynamic* type; the dynamic attribute is any attribute which can only be known at *runtime*.

The dynamic type of a reference can only be determined from the call to `new` which constructed the object to which it refers. Whatever type was constructed is the dynamic type of the object.

```
Player somePlayer;
...
somePlayer = new Player("ralph");
...
System.out.print(somePlayer);
...
```

In the above code we see that the *dynamic* type of `somePlayer` is the same as the *static* type of `somePlayer`. It is not always this easy to determine the dynamic type of a variable or parameter. Consider the code for `print`:

```
public void print(Object obj) {
    ...
    String someString = obj.toString();
    ...
}
```

¹The use of static here is not directly related to Java's `static` keyword.

What is the *dynamic* type of `obj`? It depends on what actual parameter is matched with the formal parameter `obj`. In the previous listing, `print` is called with `somePlayer`. In that case the dynamic type of `obj` is `Player` and the static type of `obj` is still `Object`.

Why do we care? Because overriding methods permits polymorphic behavior, behavior where the same code (that inside of `print`) behaves differently depending on the *dynamic* type of the objects on which it operates due to overriding of methods (to `String` in this example).

10.2 Pure Console I/O

Section 9.2 described how Java programs begin execution: the interpreter begins and looks for the **public static void** `main` method provided by the class named on the command-line. Section 7.2 describes how to use `System.out` to print information on the screen. The following program uses printing to standard output in the `main` method to implement the quintessential “Hello, World!” program². This program is similar to `HelloWorld.java` in Listing 7.2 but this version does not use FANG so it must provide its own `main` method.

```

1 /**
2  * Greeting program. Standard output for a non-FANG console I/O program
3  */
4  public class Hello {
5      /**
6       * The main program. Just prints out a greeting on standard output.
7       *
8       * @param args command-line arguments; ignored by this program
9       */
10     public static void main(String[] args) {
11         System.out.println("Hello, World!");
12     }
13 }

```

Listing 10.1: Hello: Print Generic Greeting

Notice that the method comment for `main` explicitly states that the parameter, `args` is ignored by the method. It is, in almost all cases, bad form to have unused parameters. According to a study TK referenced in *Code Complete*, there is a positive correlation between unused parameters in the parameter list and bugs in the method. `main` is the primary exception to the admonition against having unused method parameters: the `main` method signature must match that in the Java language definition in order for the Java interpreter to be able to start the program.

The next program shows two new things: `System.in` is an object much like `System.out` but for standard input (the keyboard) rather than output and `Scanner` has a constructor that takes an `InputStream`. As we first saw in Chapter 9, a `Scanner` can take apart the text in an input file. In this case the values typed on the keyboard are treated as the values found in the file.

```

1 import java.util.Scanner;
2
3 /**
4  * Greeting program. Standard output for a non-FANG console I/O program;
5  * user provides their name at a prompt.
6  */
7  public class YourNameHere {
8      /**

```

²The “Hello, World!” program is a short text program presented in almost every introduction to a computer programming language. It derives from a sample in Kernighan and Richie’s 1974 book *The C Programming Language*[KR78]. The original actually printed “hello, world” without capitals or the exclamation point. The C programming language was developed to implement the original Unix operating system (which, through many twists and turns, begat the Mac OSX and Linux operating systems).

```

9  * The main program. Prompt user for their name and print a greeting
10 * for them.
11 *
12 * @param args command-line arguments; ignored by this program
13 */
14 public static void main(String[] args) {
15     Scanner keyboard = new Scanner(System.in);
16     System.out.print("What is your name? ");
17     String userName = keyboard.next();
18     System.out.println("Hello, " + userName + "!");
19 }
20 }

```

Listing 10.2: YourNameHere: Personalized Greeting

Line 18 prints a prompt for the user. It is good form to make sure the user knows what to do next (this is important in text programs just as it is with games). Line 19 uses `Scanner.next` to read a line from the input associated with the `Scanner` called `keyboard`. Since `System.in` is, by default, associated with the keyboard, the program halts, waiting for the `Scanner` to finish reading the input. Then line 20 prints out a greeting customized for the named user. An example run of the program could look like this:

```

~/Chapter10% java YourNameHere
What is your name? Dr. Marcus Welby
Hello, Dr. !

```

There is something wrong. The program paused until the user pressed the `<Enter>` key but it only read one word. Why is that?

Looking back at line 19, the `next` method is used. By default, `next` parses the input stream into *tokens* where a token is defined as a sequence of non-whitespace characters separated by whitespace characters.

When you call `next`, the `Scanner` goes down to the input stream to which it is attached (more on how this corresponds to disk files below) and asks for some number of characters. It will read characters, skipping over all whitespace (actual space characters, tabs, and end-of-line characters; anything for which `Character.isWhitespace` returns `true`). Then, once it sees a non-whitespace character, it keeps reading characters until a whitespace character is found (and the whitespace character is *unread* so that the file read pointer will reread the character the next time the file is read).

One thing to note about using `Scanner` with the keyboard: on most operating systems the standard configuration has characters delivered to Java for reading *line by line*. That is, even though the user typed the four characters “Dr. ” as the first four characters of the line of input, Java will not see them until `<Enter>` is pressed at the end of the line.

There is no universal way to change console input from line-based to character-based mode from Java. Explaining operating system and shell specific methods is beyond the scope of this text so we will assume that console input is in line mode for the remainder of the book.

How can we change `YourNameHere` so that it prints Dr. Welby’s whole name? To read a whole line with a `Scanner` from a text file we use `nextLine`. That works with a `Scanner` wrapped around standard input just as it does when reading a file. With line 19 changed (and the program renamed to `YourWholeNameHere.java`, available in this chapter’s sample code) to use `nextLine` the earlier console session would look like this:

```

~/Chapter10% java YourWholeNameHere
What is your name? Dr. Marcus Welby
Hello, Dr. Marcus Welby!

```

InputStream Files and Collections

What is an `InputStream` and how is it different than a `File`? As we discussed in Chapter 9, a `File` object is an internal representation of a file in the local file system (typically on a hard drive or some USB storage device).

A `File` refers to a given location in the namespace of the file (the complete path name including hierarchical folder names); there may or may not be an actual file with that name. The `File` object can be queried to see if the file exists and what rights the program has to the file.

The `File` object does not refer, in any way, to the *content* of the file. It is possible to instantiate a `Scanner` with the `File` object so that the content of the named file is accessible. Under the hood, how does Java represent the content of a file?

The problem facing operating system writers is that information can come from multiple different devices: the hard drive, USB devices, DVD drives, the keyboard, etc. One of the operating system's primary functions is to simplify a program's interaction with the actual hardware.

Computer scientists want to banish the details of any given device to the operating system so that they can interact with *any* device without caring about the type of the device. Again, *abstraction* is a powerful tool with which to confront complexity.

The original Unix operating system from the early 1970s treated all devices and files as character sequences. This meant that any program which could process a character sequence could process content coming from the keyboard, the network, a disk, or even from some string stored in memory. Standard input and standard output (and the standard error channel) are, internally, just files or just character sequences.

The C programming language, developed for the Unix operating system, is a direct ancestor of modern Java and the internal view of file contents as sequences of characters is a result of that heritage.

When a `Scanner` is constructed with a `File`, the `Scanner` constructor builds a `InputStream` associated with the file. A *stream* in this case is just a sequence of characters. `InputStream` supports `read` and `close`, two of the major operations we want to perform on input text files. The `read` method returns the next character³ and `close` disassociates the stream from the content to which it is attached. The `InputStream` child class `FileInputStream` can be constructed with a `File` and gets its stream of characters from the given disk file.

Internally, the `InputStream` or one of its descendants keeps track of the file read pointer and will signal when the end of file is reached (it is possible to indicate the end of file on standard input; the exact key combination to press is operating system dependent).

Beyond String Input

Consider writing a console program which takes two integers and adds them together. The program would not be hard to write: use a `Scanner` attached to the keyboard (standard input) to read something from the user. If we used `next` or `nextLine` we would get back a `String`. Though there are ways to convert `String` values to numeric values, we want, if possible, to directly get integers from the user.

`Scanner`, as we have seen, has `nextInt`. Thus we can just call that method twice to add a pair of numbers and print the sum:

```

1 import java.util.Scanner;
2
3 /**
4  * Prompt user for two integers. Print the sum of the two integers.
5  * Program then halts.
6  */
7 public class AddTwoNumbers {
8     /**
9      * The main program. Prompt user for two integers and add them
10     * together. DOES NOT WORK AS EXPECTED!
11     *
12     * @param args command-line arguments; ignored by this program
13     */
14     public static void main(String[] args) {
15         System.out.println("AddTwoNumbers:");
16         Scanner keyboard = new Scanner(System.in);

```

³The actual return type is an `int` rather than a `byte`. The reason for this discrepancy is not relevant to the current discussion.

```

17
18     System.out.print("Number: ");
19     int first = keyboard.nextInt();
20
21     System.out.print("Number: ");
22     int second = keyboard.nextInt();
23
24     System.out.println("Sum of " + first + " + " + second + " = " +
25         first + second);
26 }
27 }

```

Listing 10.3: AddTwoNumbers

When the code is run, the output is:

```

~/Chapter10% java AddTwoNumbers
AddTwoNumbers:
Number: 100
Number: 121
Sum of 100 + 121 = 100121

```

What is wrong with the code? The sum of two 3 digit numbers should never be a 6 digit number. Looking at the code, printing the program identifier and the prompts should have no effect on the final sum. The `nextInt` calls also look right. In fact, looking at the output for a moment, “100121” = “100” + “121”. That is, looking at line 27, the `+` operator is being treated not as integer addition but as `String` concatenation.

Whenever the compiler can determine that the element to the left of a `+` is a `String`, then the plus sign means concatenation. Thus the `+` at the end of line 26 is a concatenation operator; `first` is converted to a `String` and tacked onto the end of the `String` to print. That means the `+` on line 27 is also interpreted as a concatenation operator.

We need to convince Java to do integer addition of `first + second` *before* applying the concatenation operator and building the output string. When we want to change the order of evaluation in an expression, we use parentheses. Wrapping parentheses around `(first + second)` on line 27 yields the following:

```

~/Chapter10% java AddTwoNumbersRight
AddTwoNumbersRight:
Number: 100
Number: 121
Sum of 100 + 121 = 221

```

Again, `AddTwoNumbersRight.java` is included in the chapter’s sample code. It differs from the previous listing only in the parentheses in line 27.

Sentinel-controlled Loops and User Input

The final program we will look at in this section combines a sentinel-controlled loop with reading input from the user. Looking at our previous programs, it seems that whenever we read information from standard input, we first must print out a prompt on standard output. Rather than having to remember both steps every time we require input, it would make sense to factor the two steps out into a single method which we could call whenever we wanted to read the next line from the user.

```

1 import java.util. Collections;
2 import java.util. Scanner;
3
4 public class SentinelControlledLoop {
5     /**

```

```

6   * static means there is only one; keyboard can be used in multiple
7   * methods
8   */
9   private static Scanner keyboard = new Scanner(System.in);
10
11  /**
12   * Get a line from the user. Prints the prompt on the console followed
13   * by a space. Then waits for user to enter a line and returns the
14   * full line of text to the calling method.
15   *
16   * @param prompt the prompt to print for the user.
17   *
18   * @return the line entered by the user (everything up to but not
19   * including the <return> key)
20   */
21  public static String getLine(String prompt) {
22      System.out.print(prompt);
23      System.out.print(" ");
24      return keyboard.nextLine();
25  }
26
27  /**
28   * Main program. Uses getLine to prompt user and read lines in a
29   * sentinel controlled loop. User enters the sentinel value "done"
30   * when they want to quit. All other lines are converted to upper-case
31   * and echoed back.
32   *
33   * @param args command-line arguments - ignored by this program
34   */
35  public static void main(String[] args) {
36      String line = "";
37      while (!line.equalsIgnoreCase("done")) {
38          line = getLine("Line to capitalize ('done' to finish):");
39          if (!line.equalsIgnoreCase("done")) {
40              System.out.println(line.toUpperCase());
41          }
42      }
43  }
44  }

```

Listing 10.4: SentinelControlledLoop

The method, `getLine`, defined in lines 21-25, does exactly what we want: given a prompt it shows the prompt to the user (and even appends a space on the end) and then returns the next line of text typed by the user.

Why is `keyboard` not defined inside of `getLine`? And what does line 9 actually mean? The `keyboard` `Scanner` is opened on standard input and, it is hoped, it can be used to read all user input it is necessary to read. Since we expect to call `getLine` over and over, it does not make sense to construct a new `Scanner` to read the next line and then the next line and so on. It makes much more sense to read everything from one `Scanner` which is shared by all read routines.

Since `getLine` is called from `main` and `main` is **static**, `getLine` *must* be **static**. As mentioned in Chapter 5, the **static** qualifier is infectious: methods *and* fields accessed directly from **static** methods must be **static**.

The “and fields” part is new. It explains why `keyboard` must be declared **static**. Why does line 9 have an assignment directly in the line where the field is declared? Because Java permits all fields (just like local

variables) to be initialized when they are declared. This book does not (and will not) use the `declare` and `initialize` syntax on regular fields because it can be very confusing to understand *when* the initialization takes place⁴. A `static` field must be initialized this way if the initialization is to take place before any other code in the class.

That means the call to `new` in line 9 takes place *before* `main` is called by Java. We will use the in-line initialization syntax for all `static` fields and *only* for `static` fields.

After all of that, `main` is somewhat anticlimactic. The field `line` is initialized to the empty string and so long as it is not the word “done” the loop reads a line from the user (using `getLine`) and if the line is not “done” then the line is echoed in upper case.

What happened to the DRY principle? Looking at lines 37 and 39 the same test is done twice in very rapid succession. Why? The problem is known in computer science as the “loop and a half” problem. The problem is where, in the loop, to read a value into `line`. If we read the value in first thing, as we do here, then there is a problem with the last time through the loop. After having read “done”, we should *not* execute the body of the loop. Thus there must be an `if` statement and it must have the same Boolean expression as the `while` statement.

Alternatively we could move the reading of the value from the keyboard from the beginning to the end of the loop. That would mean that the last line would be read and then the loop would cycle back to the top, the Boolean expression would return `false` and the body of the loop would not execute. The following code shows the idea:

```

35 public static void main(String[] args) {
36     String line = getLine("Line to capitalize ('done' to finish:");
37     while (!line.equalsIgnoreCase("done")) {
38         System.out.println(line.toUpperCase());
39         line = getLine("Line to capitalize ('done' to finish:");
40     }
41 }

```

Listing 10.5: SentinelControlledLoop2

We have moved the problem from the last line of input to the *first* line of input. The first time through the body of the loop what value should `line` have. That is, in line 36, what do we initialize it to? The only real answer is to duplicate the code from line 39 in line 36 as well. So we can choose to have to handle an extra half of a loop at the beginning or the end of the input.

10.3 Sorting a Collection

Pull all the hearts out of a standard deck of cards and shuffle them. How would you sort them? We want to describe the process in sufficient detail that we could convert it into a computer program. A detailed description of a method of calculation is known as an *algorithm*. An algorithm is a *finite* sequence of explicit, detailed instructions which proceed from an *initial state* through a *finite* series of intermediary states to a *final state*. An algorithm must be finite so we can write it down in a finite amount of time and the algorithm must terminate in a finite amount of time as well (hence the finite series of states requirement).

Algorithms can be expressed at various levels of abstraction. We have used a mix of a high-level description in English combined with a more formal description of most of our algorithms in Java.

The initial state for our algorithm is any arbitrary shuffled ordering of the 13 hearts. The final state is the same set of 13 cards in descending order, ace high. Consider, for a moment, how you would sort the deck. Then consider how you would *describe* how to sort the deck to an eager child. You can neither see nor touch the cards but you can tell the child what to do with them. And your rules must work for any valid initial state.

Here is a possible description at a fairly high-level in English:

⁴Non-`static` fields have values assigned just before the constructor begins execution. It is not *that* hard to explain when but it is much clearer to have all initialization of fields explicitly written in the constructor so the programmer can see the order of execution and know all starting values.

```

pick up the deck
while your hand is not empty
    find the largest card in your hand
    swap largest card with the top card in your hand
    place top card on pile on table

```

Assuming the first step in the loop works, the algorithm works (each card added to the pile is the largest remaining so the pile is built from biggest to smallest) *and* the algorithm must eventually halt (there are fewer cards in the hand each time through the loop; the hand must eventually be empty).

The only problem is the first step in the algorithm: it is too complicated. We need to describe, in explicit detail, how to find the largest card in the hand.

```

pick up the deck
while your hand is not empty
    // ----- find the largest card in your hand
    assume top card is largest
    for each card below the top
        if the current card is larger than the largest so far
            largest so far is now current card
    largest so far is largest card
    // -----
    swap largest card with the top card in your hand
    place top card on pile on table

```

So, this algorithm is now expressed in terms of looking at a card, comparing its value to that of one other card, keeping track of a location in the hand, and swapping a pair of cards. These are pretty much fundamental steps in handling a deck of cards.

This algorithm is not limited to working with cards in a deck. Given an `ArrayList` of objects, we can get any value by index, keep track of a location by keeping its index, and, using `get` and `set`, we can swap values in the `ArrayList`. The only thing that is at all challenging is knowing how to order two objects. We will look at how to do that by comparing `Integer` objects using `greater than`, comparing other objects just by comparing one of their fields, and how to use `compareTo` with `String`.

Sorting Integer Objects

```

1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 /**
5  * Initialize a literal ArrayList, print it, then sort it and print it
6  * again. The three method sort matches the example algorithm in SCG
7  * chapter 10.
8  */
9 public class SortIntegers {
10     /**
11      * Initialize a list of {@link Integer} values and sort them using an
12      * insertion sort (find largest remaining, put it in the right spot).
13      *
14      * @param args command-line arguments ignored by this program
15      */
16     public static void main(String[] args) {
17         System.out.println("SortIntegers:");
18         ArrayList<Integer> theInts =
19             new ArrayList<Integer>(Arrays.asList(9, 4, 2, 8, 3, 17, 10, 15));

```

```

20
21     System.out.println("Before:");
22     System.out.println(theInts);
23
24     sort(theInts);
25
26     System.out.println("After:");
27     System.out.println(theInts);
28 }
29
30 /**
31  * Find the index of the largest value inside of aList at or after the
32  * given starting index
33  *
34  * @param aList      reference to the list in which the index of
35  *                   the largest element is to be found
36  * @param startIndex start searching at this index in the list
37  *
38  * @return a number >= to startIndex, an index into aList; if
39  *         startIndex is out of range, will return startIndex;
40  *         otherwise will always return a valid index
41  */
42 private static int largestIndex(ArrayList<Integer> aList,
43     int startIndex) {
44     int largestNdx = startIndex;
45     for (int contenderNdx = startIndex + 1;
46         contenderNdx != aList.size();
47         ++contenderNdx) {
48         if (aList.get(contenderNdx) > aList.get(largestNdx)) {
49             largestNdx = contenderNdx;
50         }
51     }
52     return largestNdx;
53 }
54
55 /**
56  * Sort aList in descending order. Uses {@link
57  * #largestIndex(ArrayList, int)} and {@link
58  * #swap(ArrayList, int, int)} to do much of the work.
59  *
60  * @param aList the list to sort
61  */
62 private static void sort(ArrayList<Integer> aList) {
63     for (int firstUnsortedIndex = 0;
64         firstUnsortedIndex != aList.size();
65         ++firstUnsortedIndex) {
66         int largestIndex = largestIndex(aList, firstUnsortedIndex);
67         swap(aList, firstUnsortedIndex, largestIndex);
68     }
69 }
70
71 /**
72  * Swap elements aList[a] and aList[b] (using array notation). Works

```

```

73  * for all valid index value for a and b (even if they are equal).
74  * Does not do anything crafty when they are equal.
75  *
76  * @param aList list in which elements should be changed
77  * @param a      an index into aList
78  * @param b      an index into aList
79  */
80  private static void swap(ArrayList<Integer> aList, int a,
81      int b) {
82      Integer temp = aList.get(a);
83      aList.set(a, aList.get(b));
84      aList.set(b, temp);
85  }
86  }

```

Listing 10.6: SortIntegers

In `SortIntegers`, lines 19-20 initialize the variable `theInts` with a *literal list*. A literal value is one typed into the source code. The `java.util.Arrays` package contains `static` methods for working with arrays. The one used here takes an arbitrary number of arguments and returns a `List` containing the right element type. The `ArrayList` constructor can take a list and copy all the elements into the newly created `ArrayList`.

Line 24 (and line 29) print out the contents of `theInts`. Because `ArrayList` overrides `toString` in a sensible way (it calls `toString` on all the elements in the list, wrapping them in square brackets and separating them with commas), we can just pass the object to `println`. The output of this program is:

```

~/Chapter10% java SortIntegers
SortIntegers:
Before:
[9, 4, 2, 8, 3, 17, 10, 15]
After:
[17, 15, 10, 9, 8, 4, 3, 2]

```

The `sort` method, lines 64-71 follows the algorithm given above except that rather than placing numbers on the table all numbers are kept inside the array. The front part of the array is sorted (from largest to smallest) and the back part of the array remains to be sorted. Each time through the loop in `sort`, one more element from the unsorted part is moved to its right position and the sorted part of the array is extended by one. Before the loop runs at all, 0 elements are known to be in sorted order; after performing the loop n times (where n is the number of elements in the array), each iteration adding one element to the sorted region, all n elements are in sorted order.

The `sort` method uses `firstUnsortedIndex` to keep track of the index (in the list) of the first element in the unsorted region. Thus at the beginning of the method `firstUnsortedIndex` is set to 0 (all elements are unsorted). The body of the loop finds the index of largest element in the unsorted region by calling `largestIndex`. The second parameter to `largestIndex` tells the method where to start looking for the largest element; by starting at `firstUnsortedIndex` it will find the largest element in the unsorted region. Knowing the index of the largest element in the unsorted region, we just swap that value with the value first value in the unsorted region; the largest element is now in the right place and we can move `firstUnsortedIndex` forward by one.

Figure 10.2 shows the sorting of this particular list visually. The left column shows the situation just before the call to `swap` at line 69 in the `sort` method. The pink region of the list is unsorted, the triangle atop the list is the index of the first unsorted value, and the red triangle below the list is the index of the largest element in the unsorted region.

Looking from the first list to the second list, you can see that the element indexed by the red triangle was swapped with that indexed by the white triangle in the *first list diagram*. Thus the values 17 and 9 were swapped. That put 17 in its sorted place in the list; the first location is sorted, the rest of the list remaining unsorted.

The right column shows what happens inside of `largestIndex` the first time through the loop. The white triangle is the `startIndex` parameter. The red triangle represents the `largestIdx` variable; on line 46 it is

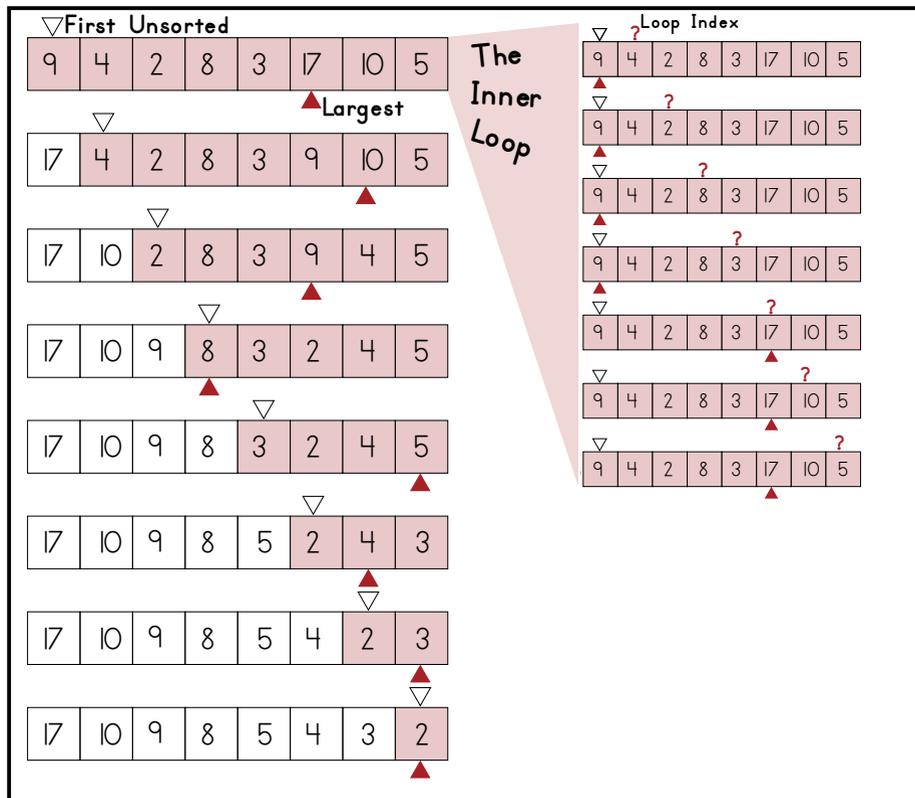


Figure 10.2: Sorting a list of Integer

initialized to be the `startIdx`. The loop control variable, `contenderIdx`, represented in the drawing by the red question mark, cycles through all the values from one more than the `startIdx` through the last valid entry. Each time through the loop, the value of the element indexed by `contenderIdx` and the value of the element indexed by `largestIdx` are compared.

If the contender is larger than the largest, largest must be updated. You can see that the red triangle (`largestIdx`) moves when 17 is compared with 9. Finally, after `contenderIdx` is past the end of the list, we know that `largestIdx` really is the index of the largest value at or to the right of `startIdx` so it is the value returned by the method (line 54).⁵

Note that in line 50 we use `>` between two `Integer` objects. Earlier it was mentioned that direct comparison between objects was not supported (and even equality tests only test for exact reference identity). Why does this compile and run? Since version Java version 1.5, `Integer` and the other object wrappers for plain-old data types automatically convert themselves to plain-old data types with the right values when Java requires plain-old data types. This is called *autounboxing* where the `Integer` object is considered a *box* around the `int`. Java also supports *autoboxing*, the conversion of a POD type to its corresponding box type when necessary.

Sorting Player Objects

How could we sort a collection full of `Player` objects if the `Player` interface were (duplicated from Section 10.1):

```
public class Player {
    public Player(String name)...
```

⁵The drawings and approach were inspired by Chapter 11, “Sorting” of John Bentley’s *Programming Pearls*, 2E. [Ben00]. The book is a collection of columns originally written for the Association of Computing Machinery’s *Communications* magazine. They are insightful, readable software engineering case studies that most computer science students can understand.

```

public String getName()...
public int getScore()...
public void takeTurn()...
public String toString()...
}

```

It depends on *how* we want to sort `Player`s. If we sort in descending order, what makes one `Player` larger than another? At the end of a game of Pig we want to sort the objects by their score. This means in `largestIndex` we will compare the result of `getScore()` called on the two elements in the list rather than directly comparing the objects. The following code is from `Pig`; the names of the methods were prefixed with `sp_` and slightly shortened so they would group together in the source code file and print more clearly. The following three methods are the last three definitions in the `Pig` class (the comments have been elided in this listing).

```

255     contenderNdx != standings.size();
256     ++contenderNdx) {
257     if (standings.get(contenderNdx).getScore() >
258         standings.get(largestNdx).getScore()) {
259         largestNdx = contenderNdx;
260     }
261 }
262 return largestNdx;
263 }
264
265 /**
266  * Sort the standings in descending order by score. firstUnsortedIndex
267  * indexes the first entry in the standings which is not yet sorted.
268  * int largestUnsortedIndex =
269  *     sp_LargestIndex(standings, firstUnsortedIndex);
270  *     sp_Swap(standings, firstUnsortedIndex, largestUnsortedIndex);
271  */
272 }
273
274 /**
275  * Swap elements standings[a] and standings[b] (using array notation).
276  * Works for all valid index value for a and b (even if they are
277  */
278 }
279 }

```

Listing 10.7: `Pig`: `sort`

Except for the change in the types and calling `aList` `standings`, `sp_Sort` and `sp_Swap` correspond exactly to `sort` and `swap` in `SortIntegers.java`. The important thing to take away from the `swap` method is that swapping two values (in a collection or not) requires a *temporary* variable to hold one of the values because as soon as a new value is assigned to a variable the old value is gone. The type of the temporary variable must match the type of the two values being swapped.

Even `sp_LargestIndex` is very similar to `largestIndex` in the integer program. Lines 264-265, where the comparison between two scores is made, is different than line 50 (in part because it is longer and therefore prints on two lines). In one case the two values returned by `get` are compared directly; in the other the values returned by `get` are used to get a particular field on which the list is to be sorted.

Sorting String Objects

`SortStrings.java` is almost identical to `SortIntegers.java`. We will look at the `main` method where the list to be sorted is declared and the `largestIndex` method where the actual comparison takes place.

```

18     ArrayList<String> someAnimals = new ArrayList<String>(Arrays.asList(
19         "dog", "cat", "hamster", "catfish", "pig", "aardvark",
20         "zebra"));
21
22     System.out.println("Before:");
23     System.out.println(someAnimals);
24
25     sort(someAnimals);
26
27     System.out.println("After:");
28     System.out.println(someAnimals);
29 }
44     int startIndex) {
45     int largestNdx = startIndex;
46     for (int contenderNdx = startIndex + 1;
47         contenderNdx != aList.size(); ++contenderNdx) {
48         if (aList.get(contenderNdx).compareTo(aList.get(largestNdx)) > 0) {
49             largestNdx = contenderNdx;
50         }
51     }
52     return largestNdx;
53 }
54
55 /**

```

Listing 10.8: SortStrings: main and largestIndex

The type of the list, `someAnimals`, is an `ArrayList` of `String`. That means that the `swap` method (not shown) has a temporary variable of type `String`. It also means that the `>` comparison we used in `SortIntegers` is not available; autounboxing only works for classes that wrap POD types and a `String` does not have a corresponding POD type.

Looking back to Section 9.4, the `compareTo` method is how objects (other than POD wrappers) should be compared. `a.compareTo(b)` returns a negative integer if `String a` comes before `b` in an ascending dictionary sort order, 0 if they are equal `String` values, and a positive integer if `a` comes after `b`. We are sorting in *descending* or reverse order so we want the largest or last in dictionary order. So if the contender compared to the largest is positive, the contender comes after the largest or, in other words, is larger than the largest. That is what line 50 does. A run of `SortStrings` looks like this:

```

~/Chapter10% java SortIntegers
SortSrrings:
Before:
[dog, cat, hamster, catfish, pig, aardvark, zebra]
After:
[zebra, pig, hamster, dog, catfish, cat, aardvark]

```

10.4 Finishing Pig

Looking back at the design of `Pig`, the program appears to ask the user for two different *kinds* of input: complete lines for the players' names and yes/no answers the rest of the time. Consider for a moment: what should the game do if the user is prompted for a yes/no answer and they answer "catfish"?

There are a couple of ways to handle this: define all answers other than the string "yes" to *mean* "no" as a default answer; make "yes" the default answer; try to deal with whatever they typed in each location where

the answer is used to make a decision; build a method which prompts the player over and over until they give an acceptable answer.

The last choice in the list has at least two advantages over the others: it uses levels of abstraction to hide complexities which are only part of getting the player to answer yes/no from any code using the answer and it centralizes all the code to handle “wrong” answers in a single place (we do not repeat ourselves).

Static Input Methods

Pig has two **static** input methods; both rely on the declaration and initialization of the **static** `keyboard` field.

```

40 // sentinel: userAnswer is a valid answer
41 while (!userAnswer.equalsIgnoreCase("y") &&
42        !userAnswer.equalsIgnoreCase("n") &&
43        !userAnswer.equalsIgnoreCase("yes") &&
44        !userAnswer.equalsIgnoreCase("no")) {
45     userAnswer = getLine(prompt);
46 }
47 // userAnswer: "y", "yes", "n", or "no"; first letter differentiates
48 return userAnswer.substring(0, 1).equalsIgnoreCase("y");
49 }
50
51 /**
61  System.out.print(" ");
62  return keyboard.nextLine();
63  */
64
65 /**

```

Listing 10.9: Pig: Input Routines

These methods are defined right before `main` in the body of the class. `getLine` is just as it was defined earlier in the chapter.

The `answersYes` method is a Boolean method which returns **true** if the user answers yes to the prompted question. The body of the method is a sentinel-controlled loop. It ignores all of the non-sentinel lines and processes the final line outside the loop so we don't have a loop-and-a-half problem.

The sentinel condition is `userAnswer` is equal to “yes” or “no” or “y” or “n” without regard to case. It is possible to write all the possible upper/lowercase mixes for “yes” and “no” (8 and 4 possibilities, respectively; generating them is left as an exercise for the interested reader) but Java provides a simpler solution. In addition to `equals`, the `String` class provides an `equalsIgnoreCase` method. It does what its name suggests: it compares two strings permitting upper and lower case versions of the same letter to evaluate as the same.

To express the sentinel condition, we need to call `equalsIgnoreCase` four times and or together the results. The sentinel-controlled loop is finished when:

```

userAnswer.equalsIgnoreCase("y") ||
userAnswer.equalsIgnoreCase("yes") ||
userAnswer.equalsIgnoreCase("n") ||
userAnswer.equalsIgnoreCase("no")

```

To use the Boolean expression of the sentinel condition in the `while` loop, we need its logical inverse. We need `while (!<sentinel>) ...` Or:

```

!(userAnswer.equalsIgnoreCase("y") ||
userAnswer.equalsIgnoreCase("yes") ||
userAnswer.equalsIgnoreCase("n") ||
userAnswer.equalsIgnoreCase("no"))

```

The expression makes sense: it is the inverse of the sentinel condition. What if having ! in front of such a long Boolean expression makes you uncomfortable. Would it be possible to use DeMorgan's rules to distribute the ! into the expression? Yes. Remember that the rules say || goes to && and each subexpression is inverted:

```
!user Answer.equalsIgnoreCase("y") &&
!user Answer.equalsIgnoreCase("yes") &&
!user Answer.equalsIgnoreCase("n") &&
!user Answer.equalsIgnoreCase("no")
```

This is exactly the expression in the `while` loop's Boolean expression on lines 43-46. Breaking it up on the operators (and keeping the parts on each line at the same level) makes it easier for a programmer to follow. It also keeps the lines short enough to print in the book.

The body of the loop prompts the user for a line of input and reads a line from keyboard. Rather than repeat that code from the body of `getLine`, the loop just calls `getLine`.

Line 50 is only reached if the sentinel condition is `true` (the inverse of the sentinel is `false`). That means `user Answer` is one of the four values. The user answered yes if it is equal to "y" or "yes". We could test for either of these with an `||` in a Boolean expression. It is also possible to note that the first character of each of the affirmative answers is "y", extract the first character, and test if it is "y". There is an argument to be made that this code is too clever (especially if it needs a comment to say what is happening) but it is clear enough that it stayed in the game.

The Main Method

The main method is the first method executed by the java interpreter.

```
74     game.getPlayersNames();
75     if (game.isPlayable()) {
76         game.play();
77     } else {
78         System.err.println("Not enough players to make game playable.");
79     }
80 }
81
82 /**
```

Listing 10.10: Pig: main

As described earlier, this method borrows from FANG in that it constructs a new object of the type `Pig` and then calls specific methods to have that object play the game. The `getPlayersNames()` method fills the collection of `Player` objects (it will be detailed in the next section). `isPlayable` is a Boolean method which checks to make sure there are enough players to play the game; at present it checks for at least 2 but it could be changed to check for at least 1 if the game should support solitaire play. Finally, if there are enough players, the `play` method is called; `play` is the game loop for `Pig` and will be detailed in the second following section.

The Player Collection

The game must keep track of a collection of `Player` objects. The field holding them is a list called `standings`. The previous section on sorting objects by a field value revealed this collection. The following listing collects together the declaration and methods (other than sorting) which manipulate `standings`:

```
21     /**
22     }
23
24     /**
25     /**
121     * Get the index of the {@link Player} p in the {@link #standings}
122
```

```

123     * list.
134         ndx = i;
135     }
136 }
137 return ndx;
138 }
139
140 /**
141  * Get player's names from the user. A sentinel-controlled loop
142  * (sentinel: when user enters "done" as a player's name) uses {@link
151     if (!playerName.equalsIgnoreCase("done")) {
152         standings.add(new Player(playerName));
153     }
154 }
155 }
156
157 /**
158  * Get input from the user to complete their turn
159  *
175 }
176
177 /**
178  * Increment the turn variable; return the next turn number.
179  *

```

Listing 10.11: Pig: The Players

The field is declared to be an `ArrayList<Player>`. The methods are in the source file (and in the listing) in alphabetical order except that the constructor comes first.

All the constructor does is initialize the field. Looking back at `main`, the next method called after the constructor is `getPlayersNames`. The method prompts the user for each player's name and adds that name to the collection until the user enters the sentinel value; this is exactly the loop-and-a-half problem. This one solves it with the repeated Boolean expression (while it could be changed to use the other method). For each name that is typed in, a new `Player` is created and added to `standings`. The newly minted `Player` has a score of 0 (we will see the `Player` class in detail below).

`isPlayable` demonstrates a useful technique for making Boolean expressions easier to read: use named subexpressions to clarify what is going on. Make an effort to separate the rules for determining, for example, if the game is playable, from how the various rule conditions are determined.

Here the game is playable if there is a player list and there are enough players. That is what line 178 says explicitly. The rule is clear and you don't need to look at lines 176 and 177 (to see how we determine if the list exists or the number is big enough) unless you need to drill down to a less abstract, more detailed level.

`getPlayer` is a convenience method. It would be possible to write `standings.get(i)` everywhere `getPlayer` is called but by wrapping it in a method it would be possible to use a different collection if we wanted to. The `indexOf` method is the *inverse* of `getPlayer`: `getPlayer` takes a small integer and converts it into a `Player`; `indexOf` takes a `Player` and converts it back into a small integer. Notice that `indexOf` handles looking up a `Player` which is *not* in `standings`. Handling that situation appropriately (returning an invalid index makes sense and matches what Java does in its collection libraries) argues that this must be a method so that the code is only written once.

`indexOf` is necessary so that we can tell when it is the first player's turn (so we can print the standings). That is part of showing the state of the game and is therefore properly part of the game loop.

Java Random Numbers

Previously we have relied on `FANG` when we needed random numbers. There are two reasons for this: the `Game` class is one you had to know in order to use `FANG` so putting random numbers in the class made them always

accessible to FANG games; under the hood FANG coordinates random number generation across multiplayer games so that every game sees the same sequence of numbers. This way when you share a dice game each Game instance, the one on each player's computer, can roll the dice itself rather than communicating the die roll from one computer to another. It makes taking a turn for any player, local or remote, identical.

We no longer have Game or any other FANG classes. How can we get a random number? Java provides a class, `java.util.Random`. An instance of the class has get

The Game Loop

The game loop itself is in the `play` method, a method which determines which `Player` is taking their turn and calls appropriate methods to show the state, get user input, and update the state of the game.

```

106     return !currHasWon;
107 }
108
109 /**
110  * Get the n'th player in the list of players.
111  *
112  */
113
114 /**
115  * Is this game playable? Are there enough players?
116  *
117  */
118
119     while (playing) {
120         curr = getPlayer(turn);
121         showState(curr);
122         handlePlayerTurn(curr);
123         playing = continueGame(curr);
124         turn = nextTurn(turn);
125     }
126     announceWinner(curr);
127 }
128
129 /**
130  * Print the standings (player's names and scores) in the order they
131  * are stored in {@link #standings}.
132  *
133  * System.out.println("Standings:");
134  * showStandings();
135  */
136
137     System.out.println();// improve readability of output
138     System.out.println(curr);
139 }
140
141 /**
142  * Find the index of the largest score inside of standings at or after
143  * the given starting index

```

Listing 10.12: Pig: Game Loop

`play` and the three main game loop methods are shown together in Listing 10.12. Again, the methods are in alphabetical order so `play` is near the middle. Each time through the loop, `curr` is set to the current `Player` by calling `getPlayer` with the turn number. That `Player` is then passed to each of the three methods to show, get input for, and update the state of the game. `showState` prints out the current player (the last two lines of the method, lines 239–240). If it is the first player's turn, it also prints out the complete standings. This is where the `indexOf` method is called; notice again the use of a name for the `boolean` expression. Thus the reason for the expression `indexOf(curr) == 0` is documented right on line 233 and the reason for the `if` statement is documented right in the next line.

The `handlePlayerTurn` method, lines 166-168, forwards the real work of getting input from the user to the `Player.takeTurn` method.

```

79  boolean rolledPig = false;
80  boolean heldPoints = false;
81  while (!rolledPig && !heldPoints) {
82      // ----- update state of turn -----
83      int roll = rollOneDie();
84      rolledPig = (roll == 1);
85      // ----- show state of turn -----
86      if (rolledPig) {
87          System.out.println(name + ": Rolled PIG!");
88      } else {
89          System.out.println(name + ": Rolled " + roll);
90      }
91      // ----- get user input for turn -----
92      if (!rolledPig) {
93          turnTotal += roll;
94          heldPoints = Pig.answersYes(name + " with " + turnTotal +
95              " points this turn, would you like to hold your points?");
96      }
97  }
98
99  // Could have gotten here for two different reasons; only update
100 // score if player held the points.
101 if (heldPoints) {
102     incrementScore(turnTotal);
103 }
104
105 System.out.println(name + " ends turn with " + score + " points.");
106 }
107
108 /**

```

Listing 10.13: `Player: takeTurn`

The `takeTurn` method looks like a game loop itself. The loop is slightly skewed in that updating the state happens at the beginning; if you rewrite it so that it is at the end you'll find this loop has a loop-and-a-half quality to it.

In the loop we roll one die and check if the player rolled a pig. Then the state of the turn is printed with the number rolled shown to the player. If they didn't roll pig then they are asked if they wish to hold. Note that lines 98-99 are a call to `Pig.answersYes`. That **public static** method isolates interaction with standard input which is why it has public visibility.

The loop finishes when the sentinel condition is reached: `rolledPig || heldPoints`. The inverse of the sentinel condition is the Boolean expression in the **while** loop.

The sentinel combines two different reasons for exiting the loop. Line 103 makes sure the score is recorded only if the player held their points. Finally, upon leaving the method it prints a short message letting the player know how their turn went.

```

117 }
118
119 /**

```

Listing 10.14: `Player: toString`

When `Pig` needs to print information about a `Player`, it passes the object to `println`. As discussed above, `println` calls the object's `toString` method. `Player` overrides `toString` to return the name and the score of the player.

After calling `handlePlayerTurn` in `play`, `Pig` calls `continueGame` with the current player. How does a game of `Pig` end? When someone wins by having a score higher than the required score. When do player's scores change? Only at the end of their own turns; it is not possible for a player to change the score of any other player. Thus we continue to play so long as the current player has not won the game. Lines 109-110 determine if the current player has won and return the inverse of that for continuing the game.

The only two things that `Pig` does that we have not looked at in detail are to print the whole standings table and to announce the winner. Printing the standings table is just looping over the contents of `standings`, passing each entry to `println` (using the `toString` override again). After the game loop ends (because someone won), the winner is announced.

```
90     System.out.println("Final Standings:");
91     showStandings();
92     System.out.println(winner.getName() + " wins with " +
93         winner.getScore() + " points.");
94 }
95
96 /**
```

Listing 10.15: `Pig`: `announceWinner`

To announce the winner `standings` is sorted in descending order by score (`sp_Sort` was examined in the previous section). Then the standings table is printed and a line identifying the winner is printed. The final line is redundant but it is important for winners to feel recognized; it improves the feel of the game for most players.

10.5 Summary

Java Templates

Chapter Review Exercises

Review Exercise 10.1

Programming Problems

Programming Problem 10.1

More Streams: Separating Programs and Data

In previous chapters we have seen how to write console programs where we can treat standard input and standard output as input and output files. We have also seen how to read information from a file so that a single game can change its behavior without being recompiled. This chapter adds the finishing touches to input and output with text files: we will see how to open output files on the disk and save information to the file, information which can be read by the same or another program.

This chapter uses the same console-based input approach found in the previous chapter so that the programs written in this chapter do not use FANG. The game in this chapter is 20 Questions; we will build a computer opponent which gets smarter and smarter the more it plays.

11.1 Outsmarting the Player: 20 Questions

The game of Twenty Questions is interesting. In various forms it dates back to TK. It is also at the heart of the recently successful 20Q TK line of portable game systems. In the two player version of the game one player chooses an object from an agreed upon domain. The other player is then permitted to ask twenty yes-or-no questions in an attempt to guess the chosen object.

For example, assume Anne and Billy are playing in the animal domain. Billy has selected a dolphin. A possible game sequence could be:

Anne: "Does it have four legs?"

Billy: "No."

Anne: "Does it have wings?"

Billy: "No."

Anne: "Is it a reptile?"

Billy: "No."

Anne: "Is it a fish?"

Billy: "No."

Anne: "Is it an amphibian?"

Billy: "No."

Anne: "Is it a mammal?"

Billy: "Yes."

Anne: "Does it swim?"

Billy: "Yes."

Anne: "Is it a whale?"

Billy: “No.”

Anne: “Is it a dolphin?”

Billy: “Yes.”

In nine questions Anne zeros in on the answer. Somewhat surprisingly, with twenty questions, if each question divides the remaining elements of the domain in half, the questioner can differentiate between $2^{20} = 1048576$ different objects. It is literally possible to find one in a million.

How can we convert Twenty Questions into a computer game? Two possibilities jump out: make a two-player game where the computer just provides a communications medium, have the computer play one of the two roles in the game. A two-player version of Twenty Questions would have the “game” acting more like an instant messaging program than an actual game; none of the rules of Twenty Questions would really end up inside of the program.

What would the program need to do to play each of the roles? Looking back at Hangman, reading a data file containing a collection of animals (or other domain objects) and selecting one at random is easy. Breaking arbitrary yes-or-no questions down and interpreting them and then answering them correctly is far beyond the scope of our current programming skills.

Alternatively, the file contains a collection of yes-or-no questions and guesses of domain objects (animals) related to each sequence of “Yes” or “No” answers provided by the user. The structure of the file is more complex than the data file we used in Hangman. Rather than a single line containing a single phrase, each item stored in the file will cover multiple values stored across multiple lines. The exact format of the data file will depend on the structure of the question and answer classes; there will be one line per *field* in the class. We now digress in the process of designing the question and answer classes to examine an interesting variation on how to read a book.

Choose Your Own Adventure

The *Choose Your Own Adventure* series was a collection of “interactive” children’s books from the 1970s and 1980s. The format of the book was that the reader began on page 1 and read the page. The page would describe the current situation and, at the bottom of the page, would provide suggested resolutions to some problem. Each resolution directed the reader to a different page in the book so the book was read non-linearly (selection rather than sequence controlled the order of the pages). The series was popular with more than 120 titles and many different imitators.

Readers of this book, forty years after the fact, should not be particularly excited by these books. Many of them have been translated onto the World Wide Web where *hypertext links* take the place of page numbers. Each of the resolutions is linked to the Web page where the outcome is described and another choice presented to the user.

Figure 11.1 is a picture of several pages from a choose your own adventure style story. Notice that each white page presents the user with a decision to make. In this snippet, each decision has only two outcomes; there is no requirement that that be true.

The pink pages have no selections on them. They represent endings of the story. Each page has a page number on the bottom; it is assumed that there are other pages in the book but none of the other pages can be reached by starting from page 1 and going only to the pages named as part of decision pages. The page numbers are widely distributed (as they were in the original series) so that it is not easy to “look ahead” by turning one or two pages ahead to see if the outcome of some branch of the adventure is good or bad for the reader.

The story in the figure, such as it is, has three possible outcomes: you die of starvation, you grow old and bitter alone, or you meet the love of your life and have a happy ending. Seeing all of the pages for all of the paths through the book makes it obvious that this book is not “interactive” in the sense that it reacts to the reader by changing due to previous actions. It is equally clear, however, that the *story*, from the point of view of a reader on any given page, is interactive. If the reader is making meaningful choices, then this is a form of a game by our working definition.

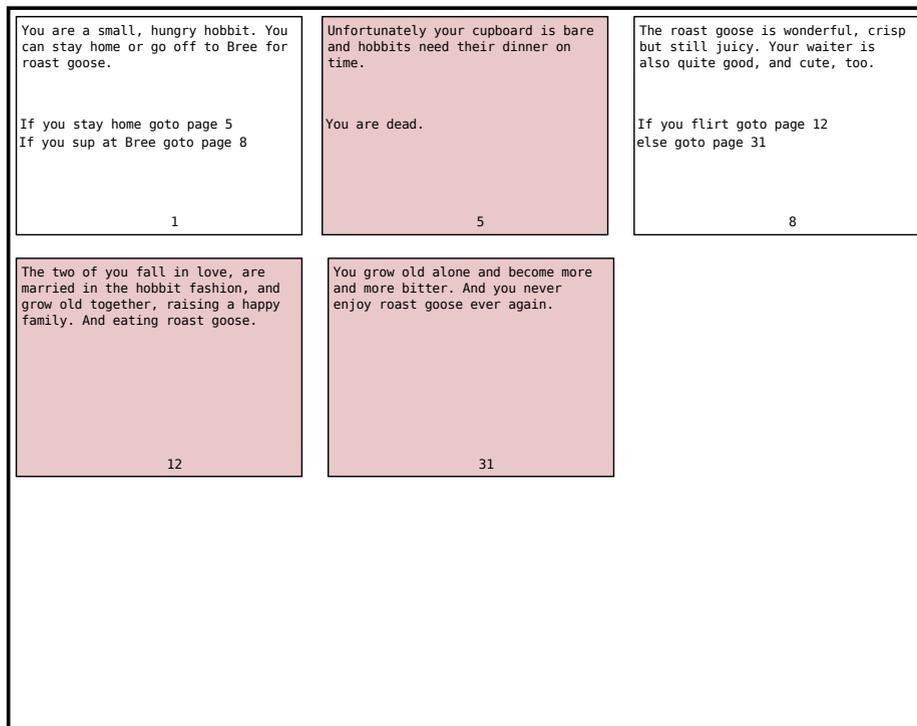


Figure 11.1: Choose Your Own Adventure example

Why are we interested in a fifty-year old literary devise¹ widely used in children’s books?

We are looking for a structure where we can store questions and answers in an easy to read manner (so we can load it from a file). The structure must also encode the relationship between the questions and the answers. That is, once the player tells us that their animal has four legs, the game should no longer guess that the animal is a dog.

The choose your own adventure book provides us with such a structure: each page is either a decision point for the user (a question) or an ending (an answer). The structure of the book is linear (the pages are printed in a fixed order in the book or stored in a fixed order in an input file) so it is easy to read. The structure of the story is encoded in the page numbers (indexes into an `ArrayList`) associated with each choice on a decision page.

Figure 11.2 is identical to the previous figure except that the text in each page has been changed. Instead of telling a story about a hobbit, the book now plays a game of twenty (or is it two?) questions. Notice that on the white pages, let’s now refer to them as `Question` pages, the two choices are now universally labeled “Yes” and “No”. Each label is then followed by a “goto page #” just as before. The pink pages, let’s refer to them as `Answer` pages, now each hold just the name of a domain object. When processing an `Answer` page, we read the text to be “Is it a(n) ...” where “...” is the name of a domain object.

To demonstrate playing the game we will pick an animal, begin at page 1, answer the questions and turn to the indicated pages. First we pick one of the animals the book knows, a turkey.

The game progresses as follows: On page 1, “Does the animal have 4 legs?”; our answer is “No” so we turn to page 8. On page 8, “Does it have wings?”; our answer is “Yes” so we turn to page 12. On page 12, “Is it a(n) turkey?”; our answer is “Yes” so the game was won by the book.

¹Raymond Queneau, a French novelist/poet pioneered the form with his 1959 *Story As You Like It* and Julio Cortázar, an Argentine author, used it in his 1964 *Hopscotch*. Both of these experimental novels had great influence on the development of hypertext and interactive fiction. See Chapter cha:TextAdventure for more.

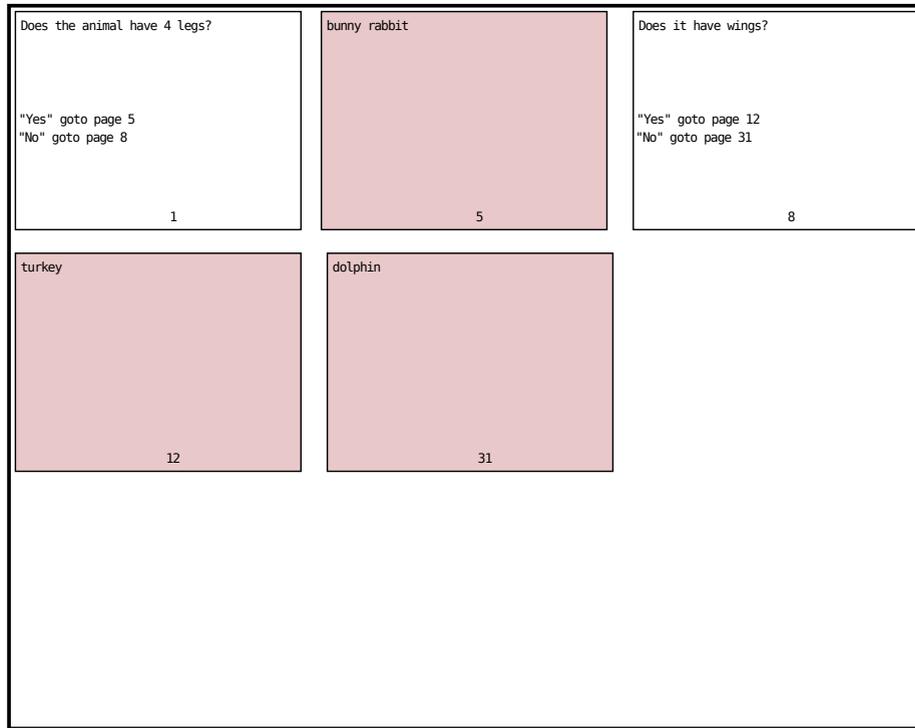


Figure 11.2: 20 Questions Book

Suppose we had selected a penguin instead of a turkey. The first two questions would go exactly as they did before. It is only on page 12, when we are asked if we had chosen a turkey, would our answer change from “Yes” to “No” and the book would be stumped.

If the book is immutable, that is its contents cannot be changed (as most printed books in the real world are), we could win every time we played with the book by picking penguin. We won once and now know we will win every time. In fact, this book is very limited and only knows three animals; all other animals stump it.

If the book is, instead, mutable, that is we can add new pages and modify existing pages, then the book can be extended, made smarter. What if when we stumped it we were asked for two things: what we had chosen and a question to tell the difference between the new and the wrong animal.

On page 12, “Is it a turkey?”; our answer is “No” so the book is stumped. “What is your object?”; “penguin”. “What yes/no question would differentiate between ‘turkey’ and ‘penguin?’”; “Does it eat fish?”. “Which answer means ‘penguin?’”; “Yes”.

Now we add two pages to the end of the book. Assuming the book had 100 pages in it before, the new pages are numbered 101 and 102. One is a question, “Does it eat fish?”, and the other is an answer, “penguin”. The “Yes” direction from the new question goes to the new answer (remember that we asked) and “No” goes to the wrong answer (page 12, remember). Finally, the question on page 8 cannot guess “turkey” when the answer is “Yes”; instead it must ask the new question.

The smarter book now knows four animals. In Figure 11.3, the updated information is written in red: the text on the new pages, the page numbers on the new pages, and the fixed up page number on the question before the wrong answer.

Another way to look at the data in the pages of our Twenty Questions books is by moving the pages around and drawing connections between the pages:

This drawing shows what computer scientists call a *tree*. A tree is a collection of nodes where one node is designated the *root* of the tree. All nodes but the root have exactly one node referring to them: the node referred to is the child of the referrer; the referrer is the parent of the referred to node. Thus page 1 is a parent

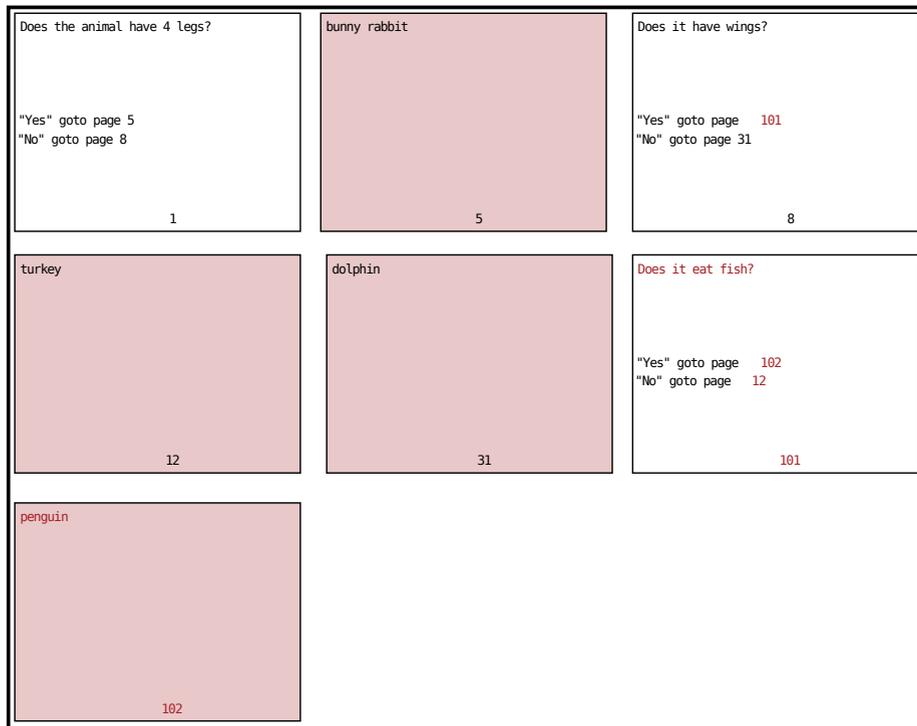


Figure 11.3: Smarter 20 Questions Book

of page 8. A consequence of having only one parent and a single root is that it is possible to traverse the tree from the root to any other node in the tree only following references from parent to child. Nodes with no children are *leaf* nodes.

Looking at Figure 11.4 you will see that computer scientists have little experience with nature: the root of the tree is at the top of the figure and the leaves are all drawn below the root.

Playing Twenty Questions with the book is really a matter of starting at the root of the tree, reading the text on the current page, and then, depending on whether the answer to the question is “Yes” or “No”, continue down the tree to the yes-child or the no-child. This is just another way of visualizing the reading of the book.

When the book loses the game, the current node is the wrong answer node. To extend the tree we need to build a small tree consisting of the new question, the wrong answer leaf and the new answer leaf, and then replace the reference to the wrong answer from the question that was its parent with a reference to the new question.

It is useful to notice that we can still get from the root to all of the nodes in the tree: since we could get to the last question in the game and we didn’t change its parent, we can still get there. Further, the last question now refers to the new question, so we can get to the new question. The new question refers to both the wrong and the new answer. Thus it is still possible to follow a path of yes and no references to get from the root of the tree to any node. The wrong answer page, in particular, has the same page number and is still reachable. The path from the root to the wrong answer goes through one more question than it did before.

Designing Our Classes

Looking at the description of a choose your own adventure story, there appear to be three classes of objects: the book, the question pages, and the answer pages. An AdventureBook is a collection of pages. Any given page can be either a question or an answer. How can we have a collection of different kinds of objects? If the objects in the collection extend the same class, the collection can have the superclass as the *static* type of its elements while the *dynamic* type of each element can be any of the specific subclasses.

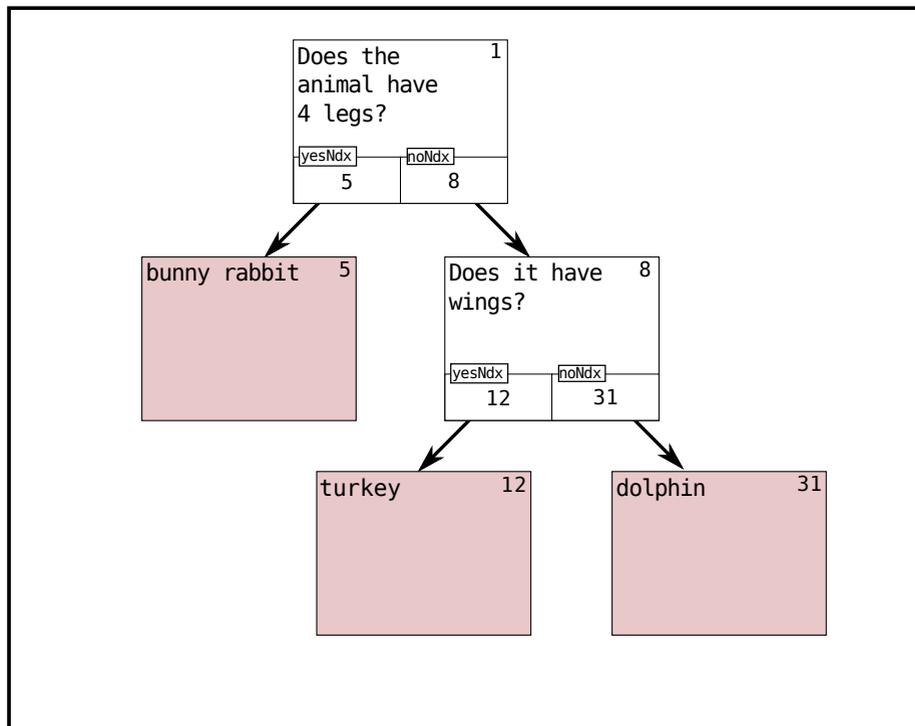


Figure 11.4: 20 Questions Tree

An AdventureBook object is a collection of AdventurePages where an AdventurePage is either a Question or an Answer. The AdventureBook is separate from the game so there is one more class, the main class, TwentyQuestions.

What does the game look like while we are playing it? Figure 11.6 shows an example. Just like in Pig in the previous chapter, the program should accept yes-or-no answers without regard to case and permitting single letter abbreviation.

It is also necessary to be able to prompt the user for an arbitrary string of input so that they can supply the name of the domain object they were thinking of and the new question text. This means that TwentyQuestions will have three **public static** methods just like Pig. The main method will construct a TwentyQuestions object and call three methods on it: `load` to load the information from the question file, `play` to actually play the game as long as the player wants to, and, optionally, `save` which will save the information known by the program back into the question file.

The call to `save` is necessary so that any new domain objects added by the player are “remembered” by the game the next time it is played. Remember the discussion of *RAM* and *disk drives* in Section 2.2. The RAM is the *working memory* of the computer. It is where the Java program, local variables, and objects constructed with `new` reside. It is volatile meaning its contents only last as long as the computer is running. It is actually worse than that in a sense: the contents of the RAM memory belonging to a Java program only last as long as the program is running.

The information about telling the difference between a turkey and a penguin must be *saved* onto the hard drive. The contents of the file on the hard drive is nonvolatile. It will remain the same until the file is deleted or overwritten. In particular, if the saved information in the file is of the right format (one we have not yet discussed), then the `load` method can be used to reload the smarter twenty questions information when the game is next played. In turn, this means that the more people play the game, the more elements are added to the collection of domain objects and the better the game becomes.

The main class’s public interface looks something like this:

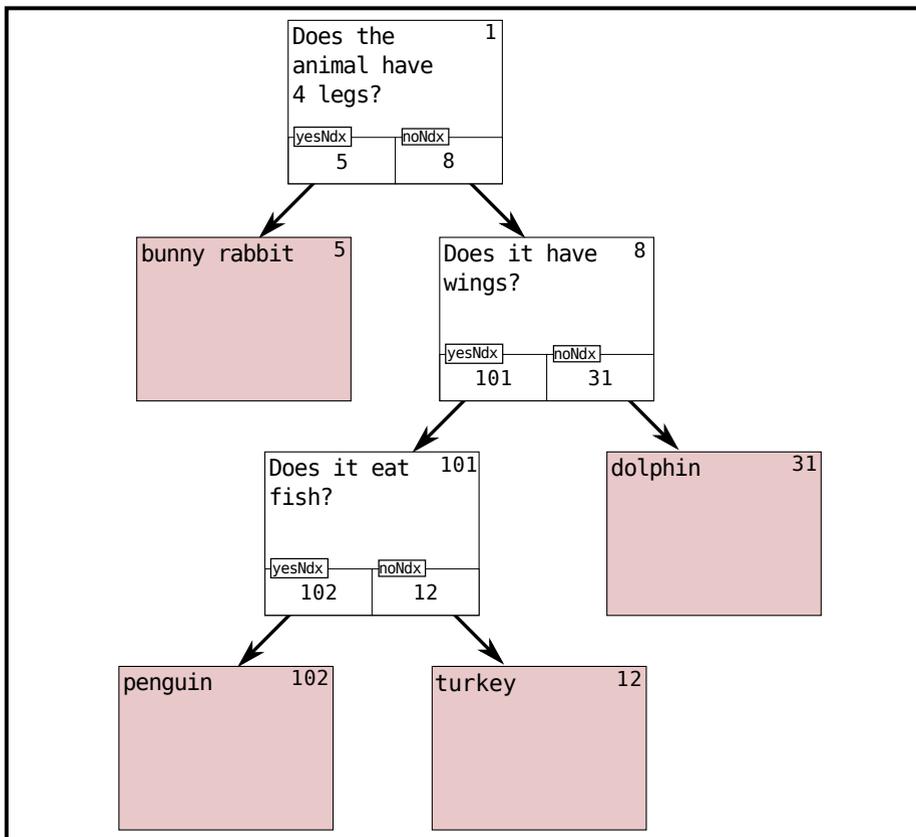


Figure 11.5: Smarter20 Questions Tree

```

public class TwentyQuestions {
    public static boolean answersYes(String prompt)...
    public static String getLine(String prompt)...
    public static void main(String[] args)...

    private TwentyQuestions()...
    private void load(String fname)...
    private void play()...
    private void save(String fname)...
  
```

The constructor and the playing methods are **private** just like those in *Pig* because they can be. They are only called from the main method *in the same class definition*. Thus they can be **private**.

```

63     String fname = args[0];
64     TwentyQuestions game = new TwentyQuestions();
65     game.load(fname);
66     game.play();
67     if (TwentyQuestions.answersYes("Save guessing data?")) {
68         game.save(fname);
69     }
70 } else {
71     System.out.println("Usage: java TwentyQuestions <fname>");
  
```

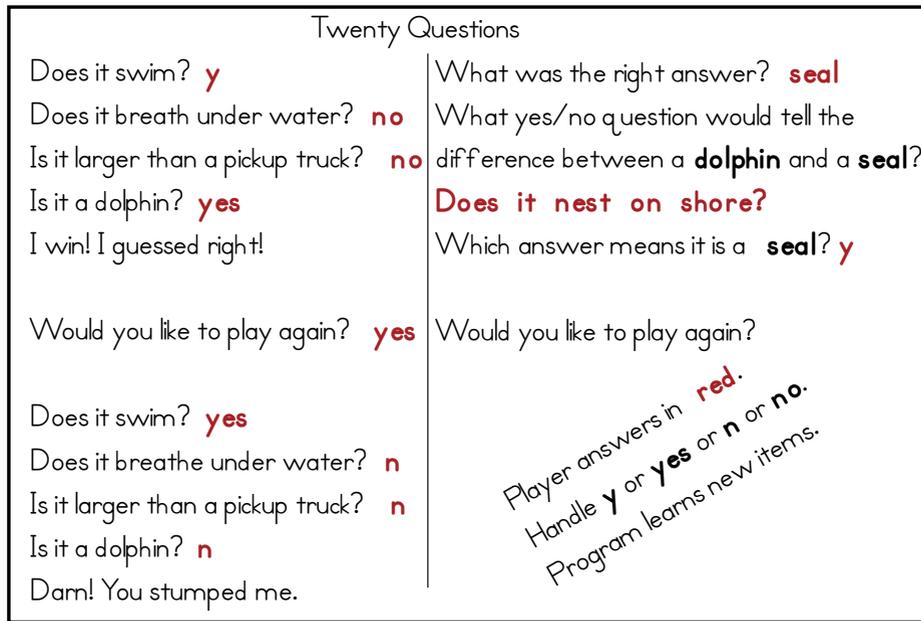


Figure 11.6: TwentyQuestions Program Design

```

72     System.out.println(" where <fname> is the data file name");
73     }
74 }
75
76 /** root of the DecisionTree representing the smarts */

```

Listing 11.1: TwentyQuestions main

Given the TwentyQuestions interface, the main method almost writes itself. The main things it does is get a file name from the user, construct a new TwentyQuestions object, and then use the load method to load the data file for the game. The play method plays the game with the user. Then, finally, if the user wants to save the changed game data, the save method is called.

The AdventureBook needs to support the same load, play, and save sort of methods. The AdventurePage class uses indexes into the book to keep track of the parent and any child nodes (in the tree), so the AdventureBook has to behave like an ArrayList and support getting an entry by index, finding the index given a page, adding a new entry, and being able to determine the size of the book overall. These requirements lead to a public interface of the form:

```

public class AdventureBook {
    public static AdventureBook readBookFromFile(String fname)...
    public static void writeBookToFile(String fname, AdventureBook book)...

    public void add(AdventurePage page)...
    public AdventurePage get(int pageNumber)...
    public int indexOf(AdventurePage page)...
    public void play()...
    public int size()...

```

Because an AdventureBook is treated like a collection, we will need to support some of the standard collection methods such as size and get as well. We will want to be able to find the page number of a given page inside the book.

Loading is done through a **static** method which loads a whole book from a named file, returning a reference to the new book. A load method like this must be **static** because it must be callable *before* any `AdventureBook` object has been constructed.

A **static** load method like this is sometimes known as a *factory*, a method which wraps up calls to **new** so that the implementation details of the constructor don't need to be known by *clients* of the class. A client is any class using objects of a given class. `TwentyQuestions` is a client of `AdventureBook`. If we consider the parameter list for a `AdventureBook` to be an implementation detail, wrapping the call to **new** inside of a factory hide the detail from `TwentyQuestions` and the programmer writing it.

The save method is also **static** to remain parallel with the loading method. It is very good practice to make sure that the parameters to methods which go together conceptually are at the same level of abstraction; it would be possible to write `writeBookToFile` to take an open file rather than a file name. That would make it hard for programmers using the class to remember which takes a file name and which takes an open file. This way the related methods are “the same” in their interface.

The two node types found in the book. `Question` and `Answer` have a lot in common. Looking at the picture of the tree, both contain text, both have a parent, and both have a page number. It is also possible to play both kinds of pages. This gives us a feeling for the public interface for `AdventurePage`:

```
public abstract class AdventurePage {
    public static AdventurePage readPageFromFile(AdventureBook book,
                                                Scanner scanner)...
    public static void writePageToFile(AdventurePage page,
                                       PrintWriter out)...

    public int getNdx()...
    public String getText()...
    public abstract boolean play()...
```

The `readPageFromFile` is another example of a factory, a method which constructs either a `Answer` or a `Question` depending on what type of page is specified in the input file (the formatting described below). This factory takes the place of a *polymorphic constructor*. Remember that a polymorphic method is a method which is overridden in subclasses. The code calls the method on an object of some *static* type but the version of the method that executes depends on the *dynamic* type of the object referred to.

Since the dynamic type of an object depends on the constructor which is called, an actual polymorphic constructor makes no sense in Java. This factory method is the next best thing. The input file connected to the `Scanner` will encode the desired dynamic type and `readPageFromFile` reads the encoding and uses an **if** statement to call `new Answer(...)` or `new Question(...)`. The factory returns a reference to the superclass `AdventurePage` but inside the method it calls the constructor of one of the subclasses of that class. More on why this is a good idea later when we discuss the file format.

What does **abstract** mean? It appears twice, once modifying the class `AdventurePage` and once modifying the method `play`. An **abstract** class is a class which contains one or more **abstract** methods. An **abstract** method is a method which provides only a signature; an **abstract** method has no method body. As a consequence of having “empty” methods, Java does not permit programmers to call constructors of **abstract** classes directly. An **abstract** class can only be instantiated by some subclass which extends it *and* overrides all **abstract** methods.

Another way to think of a **abstract** method is as one that provides only the interface. The signature of the method provides the return type, name, and parameter list which must be overridden so it describes one aspect of the class's interface. Since no implementation is provided, all non-**abstract** subclasses must provide an implementation. Thus `AdventurePage` references can be used to call `play` but it will always be the `play` method defined in a subclass.

A `Question` is a page which has two additional indexes, one to turn to if the answer to the question on the page is “Yes” and another to turn to if the answer is “No”. Turning to the next page after asking the question is embedded in the `play` method. The only other operation which needs access to the index values is `userProvidesNewAnswer`, the method which extends the tree with a new question and a new answer. The extension method is part of the interface because it is called from `play` in an `Answer` node (that is where the game will find out that it has guessed wrong). The interface for `Question` is thus:

```

public class Question
    extends AdventurePage {
    public Question(...)... // don't know signature yet
    public boolean play()...
    public String toString()...
    public void userProvidesNewAnswer (Answer wrongAnswer)...
}

```

An `Answer` is even simpler. It has to implement `play`, `constructor(s)` and `toString`. The `toString` method is in both `Question` and `Answer` so that we can print out the content of a tree and see if it is put together the way we think it is. Remember that `toString` is provided by `Object` and that it is used by `System.out.println` to convert object instances for printing. The whole `Answer` interface is thus:

```

public class Answer
    extends AdventurePage {
    public Answer(...)... // don't know signature yet
    public boolean play()...
    public String toString()...
}

```

Before we delve further into **abstract** classes and how to encode domain knowledge for a *smart* computer program, we will take time to extend our knowledge of file I/O to include file output. That way we can write text files as well as read them.

11.2 Reading and Writing Files

In the last two chapters we have seen how to open files for input using the `File` class and then read them with the `Scanner`. In the last section we saw how to use `Scanner` to read standard input; we were able to leverage what we know about reading from data files to read from standard input.

Since Chapter 7 we have, on and off, used `System.out`, a `PrintWriter` object to write text on the console. In this section we will leverage what we have seen about writing to standard output to help us write output files.

System.out and Output Files

Looking at the `System` documentation, the `public` field `out`² is a `PrintWriter`. Following the documentation link over to `PrintWriter` it is nice to see that it has a constructor which takes a `File`. It looks like we can construct a `PrintWriter` in almost the same way we construct a `Scanner` and then use the writer the same way we use `System.out`. Output will not appear in the window with the command-line but rather will be saved in a file stored in the file system.

Two important caveats. When a `PrintWriter` is constructed with the name of an existing file and you, the user who ran Java, have permission to overwrite the file's contents, any existing contents are lost. In most graphical user interfaces, when saving over an existing file, programs ask if you are sure you want to overwrite the existing file contents. You can write a program which does that but `PrintWriter` is of low enough level that when you open the file (call the object constructor), the file is reset to have a length of 0 characters waiting for you to write new contents. You must also *always* close your output files. There are file systems which will lose the contents of a file if the program terminates and an output file was not properly closed. Very few *modern* file systems will do this but it is important to make it a habit to close all files (but especially output files).

For practice writing a file, let's modify `AddTwoNumbersRight.java` so that instead of writing the sum to standard output the program instead writes it to a file. We will prompt the user for the name of a file in which

²Just a reminder: Good object-oriented design abhors *public* access for fields.

to save the results and then we will prompt the user for the two numbers to add and then we will print the results in the output file.

```

19 Scanner keyboard = new Scanner(System.in);
20
21 System.out.print("Results file name: ");
22 String fname = keyboard.nextLine();
23 File outFile = new File(fname);
24 try {
25     PrintWriter out = new PrintWriter(outFile);
26
27     System.out.print("Number: ");
28     int first = keyboard.nextInt();
29
30     System.out.print("Number: ");
31     int second = keyboard.nextInt();
32
33     // Print results in the file
34     out.println("Sum of " + first + " + " + second + " = " +
35         (first + second));
36     out.close();
37
38 } catch (FileNotFoundException e) {
39     e.printStackTrace();
40     System.err.println(
41         "Program cannot make useful progress: terminating.");
42     System.exit(1);
43 }
44 }
45 }

```

Listing 11.2: FileTwoNumbersRight

Most of main is familiar. Even the **try...catch** block is familiar; we have just not used it for `PrintWriter` objects before. Just like when creating a `Scanner`, it is possible that the file represented by the `File` object cannot be found. Typically `PrintWriter`, because it is opening the file for output, will just create a file. But it is possible that the user does not have permission to create the named file for some reason. If that happens, then an exception is raised by the `PrintWriter` constructor. This program moved all the useful code inside the **try** block because a failure to open the output file moots any values calculated by the program. Thus the exception is handled by reporting it and terminating the program.

Let's run the program:

```

~/Chapter11% java FileTwoNumbersRight
FileTwoNumbersRight:
Results file name: sum.txt
Number: 100
Number: 121

~/Chapter11% java EchoFile sum.txt
Sum of 100 + 121 = 221

```

Notice that there was no output from the program after the second number was entered. The program simply terminated. When `EchoFile` (see Listing 9.11) was run (at the next command-line prompt), it echoed the contents of the newly created output file. Java does not add any extension to a file name. The full name, `sum.txt` was provided to the `File` constructor.

When a `PrintWriter` is constructed with the name of an existing file, the contents of the file are truncated (the size is set to 0) and then anything printed to the output file are appended to the file pointer which is then moved to the end of the input.

So, if we run the program a second time with the same file name but different input, the following console session results:

```
~/Chapter11% java EchoFile sum.txt
Sum of 100 + 121 = 221

~/Chapter11% java FileTwoNumbersRight
FileTwoNumbersRight:
Results file name: sum.txt
Number: 87
Number: -5

~/Chapter11% java EchoFile sum.txt
Sum of 87 + -5 = 82
```

The first run of `EchoFile` shows that the contents of `sum.txt` are just what they were at the end of the last session. Then we run the `sum` program and finally look at the new contents. One reason for choosing the numbers that we did are because the results line is *shorter* (in characters) than the previous results line. Thus if any characters were left from the previous file contents we should see them at the end of the line after “=82”.

11.3 Data-Driven Programs

Consider a commercial game program for a moment. What makes Epic Game’s *Unreal Tournament* series of first-person shooters so popular? There are many arguments to be made in favor of gameplay decisions made by the designers along with the powerful graphics engine underlying the game. The author would argue that there is an additional factor: user-created content.

Unreal Tournament and many other popular games come with *level editors* that permit motivated players to build their own game levels. These user-designed levels can then be shared with other owners of the game. These user-generated *game mods* (short for “modifications”) give creative players a new level (pun intended) of interaction with the game and give other players a way to extend their interaction with the game in new environments. Popular mods can even go on to become their own games³.

What is a level editor? It is a program which permits users to place virtual game objects inside of virtual game spaces and to attach actions, or some sort of code, to them. The details of writing a mod or using a level editor is beyond the scope of this book since we are focused on writing games in Java, not the various modification languages of commercial game engines. What is important to us is what happens when you press the “Save” button in the level editor.

What gets saved and how does whatever gets saved become a new map, character, or game? The following discussion will assume the saved information is a new map but the general features are independent of the exact type of mod written. The level editor saves a *data file* which describes the new map. The format of the data file depends on the game engine being used; both the level editor and the game itself were written with a particular format in mind.

When the game is run, among the various command-line parameters it supports is a map file name. If you include a command-line parameter of the form `--map-file-name=MyNewMap`⁴ the game will start not in its standard first level but instead in the world described in the new map file.

The important thing to notice is that the *program*, the game itself, is not changed. The modder does not need access to the source code of the game, nor do they need to compile source code into executable code. The format saved by the level editor is the same format used by the game’s own levels.

³ *Quake* spawned *Team Fortress*, *Half-life* was modded into *Counter Strike*, *Unreal Tournament* spawned *ChaosUT* and *Tactical Ops*; these examples barely scratch the surface, focusing only on mods which became separately distributed games and only on first-person shooter games. The range of maps, new characters, weapons, gameplay modes, and the like is much, much larger.

⁴ This syntax is fictional but similar to that found in several commercial games.

Separating the code (the compiled game) from the data (the level files), the game has increased flexibility. During development the game designers can make changes without needing to compile the whole program; this is a major time savings since compiling a moderately complicated game on a single computer can take upwards of two hours. Since changes are inexpensive (in terms of time), game designers can try many different things to find the ones that work. The separation also makes it possible to sell downloadable content to extend the life of the game and provide additional revenue to the game company. And, as discussed above, separating code and data permits the user community to mod the game and even develop new games atop the existing game.

Human-readable Data Files

In the previous chapter we wrote Hangman to use an external word list file. That was a separation of data and code. It would be possible to have multiple word files, perhaps by natural language. Then you could play Hangman in German or English, Dutch or French. There could also be different files for different skill levels. If the player were asked for their school grade level, then the game could open one of a dozen different word lists and play a game tailored for the player's level.

One thing to note about the Hangman data file is its format: each line represents one entry, one phrase (or word) which the game will use to challenge the player. Knowing this is how the file is laid out, anyone could produce a data file for Hangman. This is a major advantage to using a plain text file of some format.

The next section looks at how objects more complex than a `String` can be encoded and stored in a human-readable and, more importantly, human-editable format. Knowing how the game data files are formatted, we can write `TwentyQuestions` files to guess animals (as we develop in the chapter), famous politicians, Roman trading centers, or anything else we wish.

11.4 Encoding Objects to Read or Write Them

The power of separating programs from data comes from the ability to use the exact same program to play multiple different games. The ability to play more than one game is dependent on being able to build a data structure that supports at a minimum sequence and selection. Then, once the data structure is developed as the heart of the game, it is necessary to encode it so that different structures can be saved into data files and a data file can be specified when the executable is run.

Pages and Books

In Section 11.1 we looked at how a choose your own adventure story could be stored in a book. As an analogy, a book is a good fit with a `ArrayList`: the elements (pages) are stored in sequence, each is associated with a small integer (page number), and by containing a small integer one page can refer to another page. Equally important, the reference from one page to another page does *not* require the referred to page exist when the reference is made; the page only need exist when following the reference.

This makes loading/saving a linear representation of a book simpler than it could be. If every page *referred* to by a page must have been loaded before the referring page can be loaded it would be necessary to sort the pages of the book so that leaves were in the file before any internal node that referenced them. The root of the tree would have to be the very last page read from the book file.

So, an `AdventureBook` is a linear container of `AdventurePage` objects. Looking back at the design of the interfaces of the `AdventurePage`, `Question`, and `Answer`, what fields do they each have for implementation? That is, how can the interface be implemented. We need to determine the data contained in the objects before we can determine how to store any of these objects into a text file (for reading or writing).

Let's start by agreeing to label each page in the text file with the type of the page. Thus if we have a single question with two answers on the following pages, the text file would look something like this:

```
Question
...
Answer
```

```
...
Answer
...
```

The ellipses stand for whatever information, in whatever format, is required for defining a `Question` or an `Answer`. Looking back at Figures 11.2-11.5, it is clear that every page has a page number, contains text, every page refers to a parent page, and every page belongs to some particular book.

How should these common features be written in the text file representing a book? Since the text file represents a book we can assume that the book a page belongs to is indicated, implicitly, by the file the information is in. Further, each page in the book will be represented in the file so the pages are numbered, again implicitly, from the beginning of the file. Thus the file shown above corresponds to the tree in Figure 11.7.

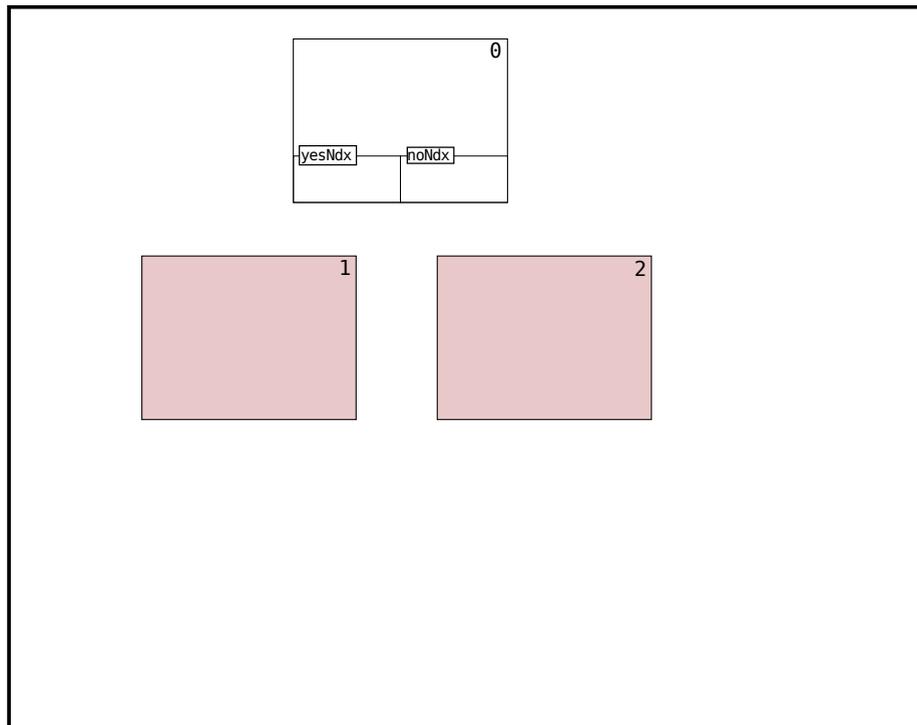


Figure 11.7: Reading a 20 Questions Tree

The tree shows the page number assigned to each of the pages when they were read in from the file: since they are stored in a Java collection it is convenient to use 0-based numbering. The layout of the page indicates the parent/child relationship between the pages but we have not yet specified the formatting for all of the fields.

In Hangman we used one line per item. Here we have committed to multiple lines. An `Answer` needs text and the page number of its parent. That sounds like two additional lines. A `Question` has the same information, text and the parent page number, as well as the page numbers for its yes and no following pages. Given a line per page number, the data file for a very simple game tree would look like this:

```
Question
Does the animal have 4 legs?
-1
1
2
Answer
```

```
bunny rabbit
0
Answer
turkey
0
```

This should be read into the following internal representation:

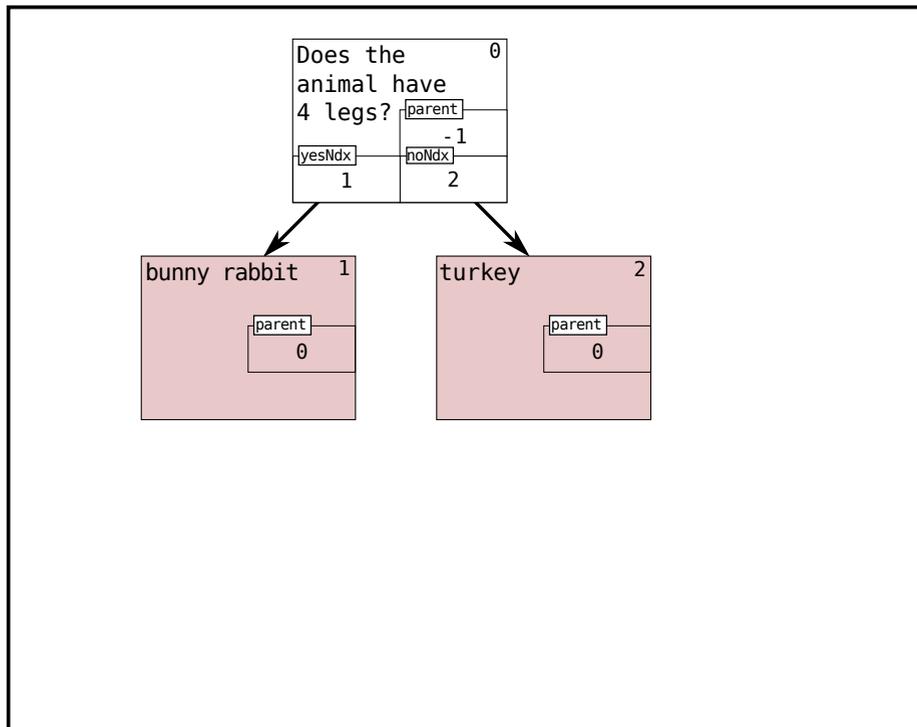


Figure 11.8: Finishing Reading a 20 Questions Tree

AdventurePage

The common information is stored in `AdventurePage` and the specialized information for `Question` is stored in that subclass. We will look at the *constructors* for the three page classes. The one thing to note is that a `Answer` or `Question` constructor is called right *after* the label is read. So the read marker is at the beginning of the text line. We will see how to arrange that when we discuss the factory method's implementation.

```

50  */
51  public static AdventurePage readPageFromFile(AdventureBook book,
52  Scanner scanner) {
53  String questionOrAnswer = scanner.nextLine();
54  if (questionOrAnswer.equals("Question")) {
55  return new Question(book, scanner);
56  } else {
57  return new Answer(book, scanner);
58  */
59  public static void writePageToFile(AdventurePage page,
60  PrintWriter out) {

```

```

71     page.save(out);
72 }
73
74 /**

```

Listing 11.3: AdventurePage constructor

The AdventurePage constructor copies the three parameters into the class's fields. The only *interesting* bit is where the page is added to the book. This makes sure that every new page is actually in the book it references as its containing book. Notice that there is no field for the page number of this page. That is because we can always use the book's `indexOf` method to find our index in the collection (which is the page number of the page). Another application of the DRY principle.

```

21 }
22
23 /**
24  * Construct an answer with the given text
25
26 /**
27  * Play this {@link Answer}: generate a guess question and ask the

```

Listing 11.4: Answer constructor

Answer has two constructors and no fields (beyond those inherited from AdventurePage). The second constructor passes its parameters to the AdventurePage constructor. The first constructor is called with a AdventureBook and a Scanner with the read pointer at the beginning of the text line. Line 22 reads the next line from the input as the text and then reads an integer (on the next line) as the index of the parent page. These values are passed directly into the other constructor.

There is a problem at the end of line 22: `nextInt` leaves the read pointer at the character following the integer's characters. That is, the read pointer is at the end of the line with the integer rather than at the beginning of the next line. This is a problem because if a call were made to `nextLine`, it would read from the read pointer to the end of the line with the integer.

We want to skip over everything up to and including an end of line character. The Scanner class has a `skip` method which takes a String describing the pattern to skip over. In the pattern, "." means any character, "*" means 0-or-more of what comes before, and the slash n is the end of line marker. Thus line 23 skips over anything up to and including an end of line marker. Just what we want.

```

18
19 /**
20  * Construct a new {@link Question} from values read from the file to
21  * which the given {@link Scanner} is attached.
22  *
23
24 /**
25  * The primary constructor for a {@link Question}. Takes the book in
26  * which the question appears, the text of the question, and the
27  * indexes of the parent, yes, and no pages for this question.
28 }
29
30 /**
31  * Play the game by asking this question. Based on user's response to
32  * the question, continue play by "turning" to the yes page or the no
33  * page.

```

Listing 11.5: Question constructor

The `Question` constructors look a lot like the `Answer` constructors: the one passed a `AdventureBook` and a `Scanner` reads all the values from the input file and passes them to the other constructor; the other constructor takes five parameters, passing three up to `AdventurePage` and setting the two local fields to the appropriate values. The two local fields are `yesNdx` and `noNdx`, the page numbers that follow this question if the answer is “Yes” or “No”.

The Factory

So, where do the `Answer` and `Question` lines in the data file come in? The factory method (the *polymorphic constructor*) can construct either a `Answer` or a `Question` from the contents at the read pointer of the `Scanner` passed into the method. The decision of which type of page to construct is made by reading the next line.

```

25  */
26  protected AdventurePage(AdventureBook book, String text,
27      int parentNdx) {
28      this.book = book;
29      book.add(this);
30      this.text = text;
31      setParentNdx(parentNdx);
32  }
33

```

Listing 11.6: `AdventurePage` `readPageFromFile`

The `readPageFromFile` method is `static` so that it can be called before any pages are actually constructed. It is passed an `AdventureBook`, to which the next page read will be added, and a `Scanner`. The `Scanner` provides the next line from the file. Since we assume the input stream is position at the beginning of a page record (either 3 or 5 lines long, beginning with the name of the class), the method reads the next line and compares it to “Question”: if it is equal, a `Question` constructor is called; otherwise an `Answer` constructor is called. Remember that the `AdventurePage` constructor (called from the subclass constructor) adds the page to the book. This makes sure that the read page is given an index (its own page number) as well as having a reference from the book.

If there were a third type of page we would have to write a new `AdventurePage` subclass and update the factory so that it could handle the new kind of page. When reading from a file it is common to use database terms: each page in the text file is a *record* composed of a type identifier and some number of *fields*.

AdventureBook

The factory can, given an open input text file, read the next page record. The constructors for the pages, as we saw above, leave the input read marker just past the end of the last line of the record. How can such a method be used to read a file?

`AdventurePage.readPageFromFile` is similar in function to `nextLine` in `Scanner`: `nextLine` reads from where the input marker is, constructing a `String` until it reaches the end of the current line; the input marker is left just past the end of line marker and the newly constructed object is returned. Thus a sentinel-controlled loop that looks much like our other file reading loops should be used to read in all of the pages for a book.

```

32
33  while (scanner.hasNextLine()) {
34      AdventurePage.readPageFromFile(this, scanner);
35  }
36  }
37
38  /**
39      AdventureBook book = null;
40      try {

```

```

52     Scanner scanner = new Scanner(file);
53     book = new AdventureBook(scanner);
54     scanner.close();
55     } catch (FileNotFoundException e) {
56         book = null; // return an error value
57     }
58     return book;
59 }
60
61 /**

```

Listing 11.7: AdventureBook readBookFromFile

The **static** `readBookFromFile` method takes a file name and opens the file for input, if possible. If it is possible, then a new `AdventureBook` is constructed using the `Scanner` attached to the input stream; if there is a problem, `null` is returned.

The important lines in the constructor are 80-82: it is an eof-controlled loop. If there is another line in the file, then there is another page. Thus the loop reads until there are no more lines; each call to the page factory consumes one page record from the input file, leaving the read marker just past the last line of the record. When the last record is read, the last line of the record is also the last line of the file so there are no more lines available. Thus the loop ends and the book is full of pages.

Writing Out Data

After playing `TwentyQuestions`, a player might have updated the data in the `AdventureBook`. How is a `AdventureBook` written out to a data file? It is done with a **static** method, `writeBookToFile`. This method is **static** to parallel the read method (it would make equal sense to make the save method non-**static**).

```

74     File file = new File(fname);
75     try {
76         PrintWriter out = new PrintWriter(file);
77         bookToSave.save(out);
78         out.close();
79     } catch (FileNotFoundException e) {
80         System.err.println("Problem opening " + fname + " for output.");
81         System.err.println("Decision tree contents unsaved.");
82     }
83 }
84
85 /**
151     AdventurePage.writePageToFile(pages.get(i), out);
152 }
153 }
154 }

```

Listing 11.8: AdventureBook writeBookToFile

When `writeBookToFile` is called, it is passed a file name and a `AdventureBook`. The method attempts to open the named file for output. If all goes well the file content is truncated and then `AdventureBook.save` is called. `save` is a count-controlled that traverses the pages list, calling `writePageToFile` with each page in pages in turn along with the open output file. When `save` finishes, `writeBookToFile` closes the output file and returns.

If there was a problem opening the named file, a message is printed and the method returns. Notice that it returns *without* having saved anything. It prints an error message giving that information to the user.

Lines 79-80 use `println` but not with `System.out`. The `System` class has three file handles, one for standard input, `in`, one for standard output, `out`, and one for standard error (or, in better English, “the standard

error channel”), `err`. Everything we have seen about using the `PrintWriter System.out` applies to using the `PrintWriter System.err`.

When running from the command-line, `out` and `err` are typically both attached to the screen for output. In some operating systems it is possible to split the output so that each goes to a different place⁵. It is good programming practice to split “regular” output from error messages so we will make sure to print output to `out` and error messages to `err`.

The `AdventurePage writePageToFile` method is also **static** to parallel the `read` method. It is passed a `PrintWriter` where the record should be written and the page to be written. It calls `save` on the page.

```

45     *                belongs
46     * @param scanner input {@link Scanner} attached to a proper input
47     *                stream
48     *
153     out.println(getText());
154     out.println(getParentNdx());
155 }
156
157 /**
158  * Set the value of the parent of the current {@link AdventureBook}.

```

Listing 11.9: `AdventurePage writePageToFile`

It is common for overriding methods in subclasses to literally extend the function of the method defined in the superclass. That is, `Question.save` does whatever `AdventurePage.save` does *and then* does whatever special stuff it does to handle `Question` objects.

`AdventurePage.save` must be called by all overriding definitions *before* they write anything to the output text file. This is why it is **protected** (subclasses must be able to call it). An `AdventurePage` is written by writing the name of the class of the page (`getClass`, defined in `Object` is *polymorphic*: it returns the dynamic class of the object) on a line, then the text on the next line, and the parent page index on the next line. Any subclass which must save more information, say `Question`, writes additional information following the base information. `Answer` has no additional information; no need to override the method in `Answer`.

```

250 }
251 }

```

Listing 11.10: `Question save`

`Question.save` overrides `AdventurePage.save`; in line 249 it calls the superclass definition. After the 3 lines written by `AdventurePage`, `Question` adds the yes and no page indexes, each on their own line. This matches the input format expected by the input factory described earlier.

11.5 Finishing the Game

The I/O of the game uses the same methods developed in the last chapter and the structure of the main game, as described above, is also very similar: a new `TwentyQuestions` object is constructed and its three methods, `load`, `play` and `save` are called in that order.

The `load` and `save` methods use the `AdventureBook` methods `readBookFromFile` and `writeBookToFile`, the methods described in the last section. All that is left to examine is the `play` method and the corresponding `play` methods in the book and page objects. As we look at `play` we will also come up against the `userProvidesNewAnswer` method, the method by which the game gets smarter. These two pieces, `play` and `learning`, are addressed in separate subsections below.

⁵The details on doing this are operating system and command-line processor specific and are beyond the scope of this book.

Recursion: Circular Invocation

Looking at Figure 11.9, how would you use the diagram to play a game of TwentyQuestions? One easy way is to take a coin and mark the page you are on. When you move the coin onto a page you ask the question. When the player answers, you just move the coin to a new page and *do the same thing again*.



Figure 11.9: Smarter 20 Questions Book (duplicate of Figure 11.3)

One key thing to notice about the decision tree view of the book: *each* Question is the root of a decision tree. That means if we know what to do at the root of the whole tree, we know what to do for each and every question.

We will examine the code from the top down, from `TwentyQuestions` down to the individual pages. Notice the separation of concerns as we go down: at `TwentyQuestions`'s level we ask if the user wants to play again (and again and ...); at the `AdventureBook` level we play one game starting at the root; at the `Question` or `Answer` level the focus is on displaying the state for the user, getting user input, and updating the state of the game⁶.

```

94  while (playing) {
95      book.play();
96      playing = TwentyQuestions.answersYes(
97          "Would you like to play again?");
98  }
99  }
100
101  /**

```

Listing 11.11: `TwentyQuestions` play

⁶Hopefully that last set of three things sounds familiar.

The `play` method in `TwentyQuestions` has the book play one game and then asks the user if they want to play again. This is a sentinel-controlled loop with an ending condition when the user says no more. It is also an example of reusing code we already wrote: since we have `answersYes`, we call it to ask if the user wants to play again, just as we call it to ask the various questions in the game.

```

19
20  /** the collection of pages in this book */
130 }
131
132 /**

```

Listing 11.12: AdventureBook play

`AdventureBook.play` forwards the call to `play` in the root page. The root page is the page with the `rootNdx` as its index. As you can see in the listing, `rootNdx` is a constant set to 0. Why name it if it is only used in one place? It cannot be a DRY consideration since typing “0” on line 131 would actually be shorter.

One reason is documentation. Line 131 with `rootNdx` as the value passed to `get` indicates *why* we want that particular page. The other reason is that it would be possible to modify the file format to include not just a list of pages but also to include the root page index. We could then load the root page index from the file (it would no longer be a constant value) and the code would still work. It might be convenient to be able to rearrange the book without worrying about moving the root.

Finally we look at the `play` method in the two page classes. Remember, `play` is **abstract** in `AdventurePage`; it has *no* definition in that class so there must be overrides in all concrete (non-**abstract**) subclasses of `AdventurePage`.

```

69  if (TwentyQuestions.answersYes(getText())) {
70      nextPageNdx = getYesNdx();
71  } else {
72      nextPageNdx = getNoNdx();
73  }
74  AdventurePage nextPage = getBook().get(nextPageNdx);
75  return nextPage.play();
76  }
77
78  /**

```

Listing 11.13: Question play

In `Question` we see the complete game loop for `TwentyQuestions`: the call to `answersYes` displays the question in the text (shows the state) and waits for an answer from the user (gets user input). Based on the input (was it “Yes” or “No”) a next page is selected and `play` is called on that page (state is updated *and* the loop starts over).

Two related things to notice: there is no explicit loop construct here, instead `play` calls `play` to start the loop over again. Each iteration of the loop is one call to `play` and the method will call itself if another iteration is needed.

A method which calls itself (same name, same signature, same class of object) is a *recursive* method. Recursion is an alternative way to implement iteration. For some data structures, in particular trees and other linked-together structures (graphs), recursion can be conceptually simpler than writing a loop. Here we recur to `play` at the next question after the current question (think about moving the coin and asking the question). Then, when the user reaches an `Answer`, the `play` method on `Answer` does not recur so it returns **true** or **false** so the `Question.play` that calls it returns that value to the `Question.play` that called it which returns that value to the `Question.play` that called it which.... You get the idea.

```

48  if (TwentyQuestions.answersYes(guess)) {
49      System.out.println("I am so smart! I win!!!");
50      return true;

```

```

51     } else {
52         System.out.println("Darn, you stumped me.");
53         getParent().userProvidesNewAnswer(this);
54         return false;
55     }
56 }
57
58 /**

```

Listing 11.14: Answer play

Answer.play builds a question out of the item to be guessed and ask it of the user (again, show state, get input from user). If the player answers “Yes” we can just return **true** because we won.

If the player answers “No”, then we don’t know enough domain objects. The player has one we don’t know. So, lets get the parent of the wrong answer (it is a Question) and have it fix up the contents of the tree. We then return **false** because we lost the game.

A Learning System

Section 11.1 described how to fix up the decision tree/book when it failed to guess a domain object. That description is reflected in the userProvidesNewAnswer method. The method is longer than any other we have seen and it has more internal comments than we have seen since NewtonsApple. It is not difficult to follow but it is a little bit complicated.

```

109 String correctAnswerText = TwentyQuestions.getLine(
110     "What were you thinking of?");
111
112     // And a question to differentiate right from wrong
113 String differentiatingQuestionText = TwentyQuestions.getLine(
114     "What question could differentiate between a \" +
115     wrongAnswer.getText() + "\" and a \"" + correctAnswerText +
116     "\"?");
117
118     // And whether "yes" means the new object or not
119 boolean newAnswerIsYes = TwentyQuestions.answersYes(
120     "Which answer means \"" + correctAnswerText +
121     "\" is the right answer?");
122
123     // Make a new answer and a new question. Attach the two answers,
124     // right and wrong, to the yes/no or no/yes positions of the new
125     // question (according to what the player told us)
126 Answer correctAnswer = new Answer(getBook(), correctAnswerText,
127     NO_SUCH_PAGE);
128
129 Answer newYes = null;
130 Answer newNo = null;
131 if (newAnswerIsYes) {
132     newYes = correctAnswer;
133     newNo = wrongAnswer;
134 } else {
135     newYes = wrongAnswer;
136     newNo = correctAnswer;
137 }
138

```

```

139     Question differentiatingQuestion = new Question(getBook(),
140         differentiatingQuestionText, getNdx(), newYes.getNdx(),
141         newNo.getNdx());
142
143     // Fix up the last question so instead of guessing wrongAnswer, we
144     // ask differentiatingQuestion. Need to know whether the wrongAnswer
145     // came from the yes or the no branch of the last question
146     if (isYesChild(wrongAnswer.getNdx())) {
147         setYesNdx(differentiatingQuestion.getNdx());
148     } else {
149         setNoNdx(differentiatingQuestion.getNdx());
150     }
151 }
152
153 /**

```

Listing 11.15: Question userProvidesNewAnswer

The method asks the user for the name of their new domain object, a question to differentiate between the new and the wrong objects, and how to answer the question to mean the new object.

It then builds a new `Answer` object without a parent (the parent will be fixed up when the new question is constructed). Then the answers, new and wrong, are matched with yes and no answers so that the differentiating question can be constructed. Lines 141-143 make the new question, adding it to the book, and fixing up the parent fields of the two `Answer` objects referenced. `setYesNdx` is listed here; `setNoNdx` is parallel.

```

231     if ((yesNdx != NO_SUCH_PAGE) && (yesNdx < getBook().size())) {
232         getBook().get(yesNdx).setParentNdx(getNdx());
233     }
234 }
235
236 /**

```

Listing 11.16: Question setYesNdx

When the `yesNdx` in the `differentiatingQuestion` is set, the `parentNdx` of the page (since both `Answer` objects are already in the book so they already have valid indexes) referenced is set to the current page.

Finally, the current `Question` must be updated. Why? Because this `Question` refers to the wrong answer (we called the method on the wrong answer's parent, remember?). It must refer, instead, to the `differentiatingQuestion`. Only difficulty: do we fix up the `yesNdx` or the `noNdx`? Depends on whether the wrong answer's index is in the `yesNdx` or not. `isYesChild` returns `true` if the given index is the `yesNdx` (again, a named method to explain *why* we compare the two indexes).

We will look at one more method, the `getParent` method in `AdventurePage`. The method is fairly simple in concept: go to the book, get the object with the parent's index and return that object. The problem? The pages in the book are `AdventurePages` and all parents must be `Questions`. We could just return `AdventurePage` objects and leave it to the routine that got it to figure out that it is a `Question` and figure out how to change the reference. That is not a good choice because it requires a lot of error checking all through the code. It makes more sense to make the change in one place, especially since we know that in a properly constructed decision tree any parent *must*, of necessity, be a `Question` page. We can avoid any error checking since we know what type of `AdventurePage` it must be.

```

122     protected Question getParent() {
123         if (getParentNdx() != NO_SUCH_PAGE) {
124             return (Question) getBook().get(getParentNdx());
125         } else {
126             return null;

```

```
127     }  
128 }
```

Listing 11.17: AdventurePage getParent

Line 124 uses `get` on the `book`; that returns a `AdventurePage`. The `(Question)` in front of the expression is a *cast operator*. We saw type casting briefly in Chapter 6. The form of a cast operator is the name of a class inside a pair of parentheses. If the expression following is a `Question`, then the whole expression is a reference to that `Question`. If, perchance, the expression is *not* actually a `Question`, then the cast fails and the result is `null`.

Most object-oriented programs should not need casts. In fact, this program needs casts because we use numeric references into the `pages` list rather than Java references to hook questions to answers⁷. This one cast is unfortunate but it does not completely invalidate the class structure of the program.

There are a handful of helper methods which were not described in detail in the chapter. This is because simple getter and setter methods should be easy to understand. The choice of `private` or some other access should also be understandable: choose `private` unless, for some reason, you cannot.

11.6 Summary

Java Templates

Chapter Review Exercises

Review Exercise 11.1 file to tree

Review Exercise 11.2 tree to file

Programming Problems

Programming Problem 11.1 TK

⁷That leads to recursive data structures where a `Question` contains a reference to another `Question` and requires recursive methods for reading and writing the structures to text files. For the cost of one cast the read and write routines are must easier to understand.

Lists of Lists and Collision Detection

We have spent some time working with non-FANG games, looking at running our own game loops and manipulating files. In this chapter we will go back and write a more complex FANG program. The game takes advantage of the animation capabilities of FANG and explores different ways of modeling the internal structure of the game world at the same time. Since this program is longer than previous programs, this program takes advantage of a Java structuring mechanism, **packages** to permit us, as programmers, to focus on one portion of the program at a time. `BlockDrop` also uses the factory pattern, multi-dimensional `ArrayLists`, and file input/output.

12.1 Designing `BlockDrop`

Just about everyone reading this book has, sometime in the last quarter century, played the game of *Tetris*. The game defined the “casual games” genre and has been ported, legally and illegally, to almost every game-capable platform ever developed. In this chapter we will be developing our own port¹ in FANG. In respect of the *Tetris* trademarks, our program will be called `BlockDrop`.

The design of this game will begin with a screenshot of the finished game in action; while this might seem backwards, you can just as easily imagine the screenshot is of any *Tetris* clone and we are examining the elements and gameplay in an attempt to duplicate the game. If you plan on duplicating a game for anything other than learning and personal gratification, make sure you understand trademark law in your jurisdiction.

The key things to note in Figure 12.1 are the playing board in the middle of the screen where the different shaped tetraminoes (groups of four blocks) fall from the top. The pieces are steered and rotated as they fall with various keys. When a piece can no longer fall, the block freezes on the screen. Whenever a row is filled from one side to the other with blocks from one or more pieces, that row is removed and the player’s score is increased by 1. The game ends when the top row is not empty (some portion of a frozen piece is in the top row).

The score, shown in the upper left corner of the screen, tracks the number of rows the player has eliminated. The box on the left of the screen is the “next piece box” showing the next piece that will fall.

The tetraminoes are shown in Figure 12.2 along with their common letter designations.

Think about your favorite arcade or console game. What does the game do when no one is playing it? It shows a gameplay video or it plays itself in a level of the game or it shows the high scores. This is known as an *attract screen*, something which attracts the player to the game so they press the appropriate start button.

¹In computing, to *port* is to rebuild a software system to run on a different computing platform; the resulting rewritten software is referred to as a *port*. *Tetris* has been widely *ported* to different operating systems and computers since it was first developed.

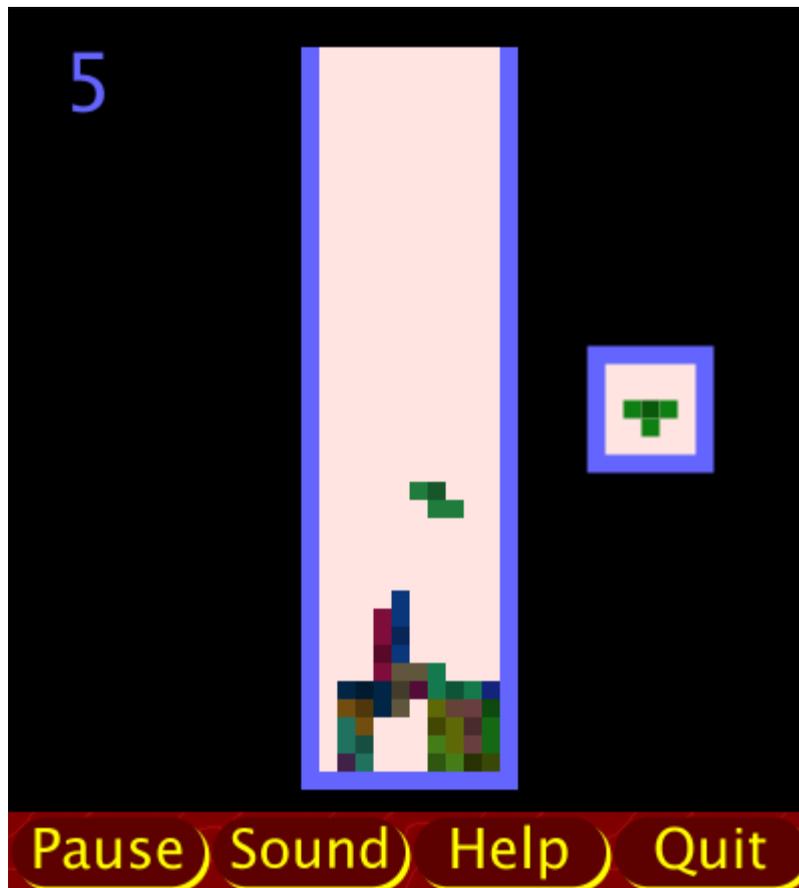


Figure 12.1: *BlockDrop* screenshot

The attract screen for `BlockDrop` will be a scrolling display of the ten highest scores earned in the game, challenging players to try their luck at getting on the leader board. In order to keep the high scores from one session of the game to the next we will need to have a high score file. The `HighScoreLevel` and `BlockDrop` will alternate, as one ends it creates and starts the other.

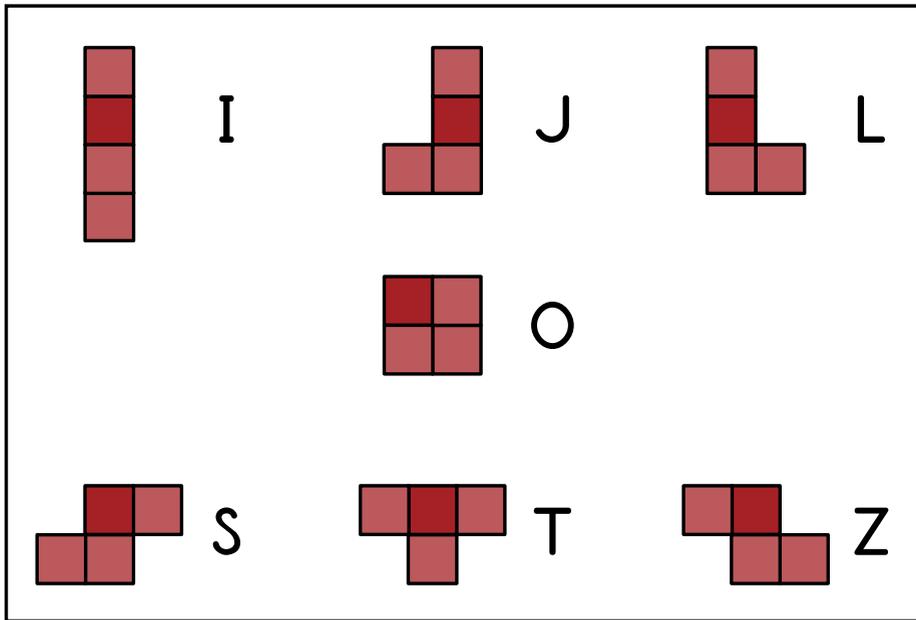
With more than ten new classes (the two `Game`-extending classes discussed above and at least nine `Sprite`-extending classes), this is the longest game with the most files we have yet examined. How can we tame the complexity? The next section will take a time out from building the game to discuss software engineering, a computer science discipline focused on efficiently writing correct, effective code.

12.2 Software Engineering: Managing Complexity

One recurring theme of this book is *compartmentalization*: write short, single-purpose methods; design classes and interactions using only the public interface; hide all fields and as many methods as possible by making them `private`. Two different impulses come together to drive this theme: to use object-oriented programming techniques well and to control complexity.

Object-oriented programming developed out of the study of *abstract data types* [ADTs]. The creation of languages with programmer-defined *classes*, *inheritance*, *polymorphism*, and *encapsulation* created the object-oriented programming paradigm.

As we saw in Section 6.3, an abstract data type has three features: a *public* interface, a *private* implementation, and interaction limited to the interface. The last feature means that *clients* of the abstract data type

Figure 12.2: *BlockDrop* Tetraminoes

cannot make use of any aspects of the private implementation. The separation of interface from implementation means that the implementation can be changed without having to change client classes.

Notice that encapsulation, the ability to hide data and other details inside a class, done in Java with **private** members and methods, comes directly from the last feature of ADTs². Java access modifiers in general are used to separate the public interface from the private implementation. **protected** elements kind of cross the border between implementation and interface: subclasses of a class with protected elements have interaction with the *implementation* of the base class. This is sometimes necessary for efficiency but it is, strictly speaking, a sharing of the implementation between the super and subclasses.

Inheritance is the ability to extend an existing class, providing only that state and those behaviors which the new class needs. The ability to override existing behavior and have the overriding definitions used through references to the original class is polymorphism. Our use of classes, inheritance, and single-purpose methods fits very well into using object-oriented programming techniques well.

The other reason for the approach used in this book is to overcome complexity. Computer programs are complex structures. Because they have no physical manifestation, there is no natural limit on *how* complex. A steel bridge can only be so complicated before it literally falls under its own weight; a computer program is only limited by the memory space within the computer.

A disciplined approach to building software is important to overcome the complexity and produce correct software in an acceptable amount of time. The study of how to do this is *software engineering*. Software engineering, generally considered a sub-discipline of computer science, is focused on quantifying software development costs, risks, and quality. Then, armed with quantitative analysis of existing projects, different approaches can be applied in an attempt to improve the metrics.

Among the practices studied in software engineering are program design patterns like our factory, the separation of interfaces and class implementations, grouping classes together into larger *modules*, and the reuse of existing code. This section will present each of these ideas and discuss how it applies to our current project.

²ADTs predate object-oriented programming; thus we talk about how ADT concepts drove the development of object-oriented programming.

Java interfaces

Every time we needed a new kind of “thing” in a program we have created a new `class` in a new file. In defining a `class` we provide the public interface and the private implementation. What if we relaxed the two requirements: that we must provide an implementation whenever we define an interface and that we must start a new file for each new `class`. This section examine interfaces without implementations. The next will look at packing more than one class in a single file.

Consider building a generic sorting class. The class should take an `ArrayList` of sortable objects, objects which implement the `compareTo` method, and, using `compareTo` and something like the sorting code we saw previously, it could *polymorphically* compare whatever actual objects were in the list, sorting the list.

We need a “superclass” that collects together all objects defining `compareTo`. Then all we need to do is make sure any new “sortable object” just **extends** the “superclass”. You can guess from the liberal use of scare quotes that there is a problem in declaring such a “superclass”. There is.

Java has *single inheritance*: a given `class` can extend one other class. Every class, except for `java.lang.Object`, *must* extend exactly one other class. This means that the inheritance graph³ is a *tree*, similar to the decision tree we saw in Chapter 11. While each node can have an arbitrary number of children, no node can have more than one parent.

If we defined a superclass for all sortable objects, then all such objects would be in the tree below that superclass. What if we wanted a `Sprite`-derived class that was also sortable? If our `SortableSprite` class extends `Sprite` and the sortable superclass; since the inheritance graph is a tree, either `Sprite` is a superclass of all sortable objects or sortable objects is a superclass of `Sprite`.

The first outcome would require all Java objects that are sortable to extend a class from our third-party framework⁴. This means Sun has to include FANG with every Java installation which makes no sense. Thus `Sprite` must extend `sortable`.

But wait, that means *all* `Sprite`-derived classes are sortable, even if that makes no sense for many `Sprite` classes. How should `LineStyle` objects compare, one to another?

What we need is either *multiple inheritance* or some way to define interfaces. Multiple inheritance is the solution in some languages such as C++ and Eiffel. It is not without cost, particularly in changing a reference to a subclass back into a reference to one of its superclasses⁵.

Java’s solution is the use of **interfaces**. An **interface** is a *totally abstract* class. That is, where an **abstract** class can have methods, declared **abstract**, which have no implementation, an **interface** has only methods which have no implementation. An **interface** is declared just like a `class`: it is declared to be **public** so that it is visible to other classes; it is declared in a file with the same name as the **interface**; the definition of the **interface** goes in a block after the name of the interface.

When writing a `class` which *implements* the methods named in the **interface**, the keyword **implements** goes between the name of the implementing class and the beginning of the class definition block. A class **extends** exactly *one* class so only one class name appears after **extends**; any class **implements** an arbitrary number of interfaces so a comma-separated list with all the interfaces it implements can appear.

We will not go into detail here but there are a large number of **interfaces** provided by Sun’s Swing graphical user interface code. FANG defines a class called a `GameWindow`:

```
public abstract class GameWindow
    extends JApplet
    implements ActionListener,
        WindowStateListener {
```

³A mathematical *graph* is a set of *nodes* which are joined into ordered pairs by a set of *edges*. The *inheritance graph* has classes as nodes and there is an edge between two classes if one directly extends the other.

⁴In a contract for software, let’s say Java, the provider, Sun Microsystems, is the *first-party*. The end-user of the software, you, the programmer of the language, is the *second-party*. FANG, provided to enhance Java but created by neither Sun nor you, is a *third-party* product.

⁵The details are beyond the scope of this book.

The above code, taken from `GameWindow.java` shows how the `GameWindow` class **extends** something called a `JApplet`⁶. The `GameWindow` also **implements** two interfaces, `ActionListener` and `WindowStateListener`. Looking at the **import** statements and Java documentation (Swing is a standard, first-party library for Java), both of these interfaces are defined in the `javax.swing` package. We will stop here with `GameWindow` and change our view to a class for `BlockDrop`, `HighScore`. We take a slight diversion from **interface** to motivate the idea of nested classes; the code for `HighScore` **implements** an interface, giving us an example of how to do that.

Nested Classes

Between games of `BlockDrop` we will display the high scores so far achieved. That means that there is an end-of-game level, `HighScoreLevel`. The main game, `BlockDrop` needs to be able to communicate the name and score for a new high scoring player but otherwise it knows nothing about high scores.

None of the visible elements discussed above in the game design know anything about high scores either. The `HighScoreLevel` must be able to keep a list of some number (say 10) high scores which group a name and a score, save the list to a file, load the list from a file, and sort the list. Note that the methods to do these things are *implementation details*. So, it turns out, is the existence of a class to group a name with a score. That is, the `HighScore` class, the element type in the list of high scores, is an implementation detail of the `HighScoreLevel`. All the fields and methods that are implementation details are encapsulated, hidden, **private** inside the class. `HighScore` should be, too.

Java supports *nested classes*, the inclusion of a **class** within the body of another class. A nested class can be given any Java access modifier so it is possible to declare nested classes to be **public** to include them in the public interface of the class or **private** to use them solely in the implementation⁷. The following listing shows the skeleton of the declaration of `HighScoreLevel.HighScore`. The **class** lines and beginning of the definition block and the closing curly braces of the definition block have been included.

```

23  /** the default score file */
24  private static final String DEFAULT_SCORE_FILENAME =
286  /** number of characters used in the full-width name */
287  private static final int SIGNIFICANT_CHARACTERS = 10;

```

Listing 12.1: `HighScoreLevel.HighScore`

Notice that the name of the nested **class** is `HighScoreLevel.HighScore`. This is similar to the naming of **static** methods and fields. We will tend to use the short name, `HighScore` when no confusion will result.

The other thing to notice in this listing is that `HighScore` **implements** an **interface**. Looking at the documentation and the code provided by Java, the definition of the interface is:

```

public interface Comparable<T> {
    public int compareTo(T o);
}

```

The first thing to do is ignore `T` for the moment. If we just let `T` be some type, then we see that this looks like a **class** but with the word **interface** in place of **class**. It has a method in it, `compareTo`. Just like when a concrete class, one on which **new** can be called, **extends** an **abstract** class, any class which **implements** `Comparable` must have a method with the given signature. An interface defines one or more signatures for methods which implementing classes must define.

The `<T>` is an example of Java *generics*. We have been using Java generics with `ArrayList` for several chapters now. The definition of the **interface** simply says that the interface is generic, taking the name of a type between angle brackets. It also says that the parameter type of `compareTo` must be the same type as that in the generic angle brackets. Defining generic classes is beyond the scope of this book so we will now return to implementing `Comparable<HighScore>` (because `HighScore` values can be compared to other `HighScore` values).

⁶A `JApplet` extends `Applet` and that is why FANG games can be run as applets inside of properly configured browsers. When they run as applications, FANG arranges to start the applet by calling `runAsApplication` (might be better to call that method `startSuperSimpleAppletBrowser`).

⁷Or **protected**, again so that they are mostly private but available to subclasses.

In order for `HighScoreLevel.HighScore` to implement the `Comparable` **interface**, it must override the required method with one having the right signature *and* a method definition.

```

340     return 1;
341   }
342   if (name.isEmpty()) {
343     return -1;
344   }
345   if (score == rhs.score) {
346     return name.compareTo(rhs.name);
347   }
348   return score - rhs.score;
349 }
350
351 /**

```

Listing 12.2: `HighScoreLevel.HighScore` `compareTo`

Tracing the code in `compareTo` we see that empty names always come after non-empty names. If neither name field is empty, then the score values are compared. If the score values are the same, then the `HighScore` objects compare in order by their name values (which means if they have the same score and name, they compare as equal which is what we would expect). If the scores are different, they sort in order by the score.

Remember that `compareTo` only guarantees that the *sign* of the return value will reflect if **this** or `rhs` sorts first (negative means **this** is first; positive means `rhs` first). Thus we don't have to use an `if` here, we just subtract the `rhs.score` from the **this**.score; if `rhs` is bigger, then the result is negative (**this** < `rhs`) and if **this** is bigger, result is positive (**this** > `rhs`).

Note that the signature of a method does not include the *names* of the formal parameters in the parameter list. While the **interface** calls the parameter `o`, when overriding the method any valid Java variable name can be used. `rhs`, standing for “right-hand side” was used to bring to mind the comparison operators shown in parentheses in the previous paragraph. **this** is on the left-hand side and the parameter is on the right-hand side. Signatures match up on the number and type order of parameters. Notice that the `@Override` annotation is used here because we are overriding the *empty* definition of the **interface**.

One other thing to note: just as any class which directly or indirectly **extends** a superclass *is-a* superclass object, any class which directly or indirectly **implements** an interface *is-a* object of the interface type. `HighScore` is-a `Comparable<HighScore>`. We can use methods which depend solely on the methods described in an **interface** and we can even write our own methods that do. The next section will show how we can use Java defined sorting methods to sort a collection of `Comparable` objects; an `ArrayList<HighScore>` is a collection of `Comparable` objects⁸.

Don't Reinvent the Wheel

Let's take a quick look at how `HighScoreLevel` holds the high scores:

```

41
42  /** game state variable: are we waiting for user to enter name? */

```

Listing 12.3: `HighScoreLevel` `highScores`

The `ArrayList` containing the scores is a field called `highScores`. When a `HighScoreLevel` object is constructed, the high scores are all loaded from a file; when a new high score is recorded, the high scores are saved back to the file from which they were loaded:

⁸`Comparable<T>` and `Comparable` are basically the same thing; the book will use `Comparable` when talking about the interface in general (any possible instance of `T`) and `Comparable<HighScore>` (or similar) when the type would make a difference. The sort routines are generic, handling any type which can be compared to itself.

```

216 File highScoreFile = new File(fname);
217 try {
218     Scanner highScoreInput = new Scanner(highScoreFile);
219     while (highScoreInput.hasNextInt()) {
220         hs.add(new HighScore(highScoreInput));
221     }
222     highScoreInput.close();
223 } catch (FileNotFoundException e) {
224     // It is okay if there is no such file; we will just ignore this
225 }
226 return hs;
227 }
228
229 /**
258     PrintWriter saveFileWriter;
259     try {
260         saveFileWriter = new PrintWriter(saveFile);
261         for (int i = 0; i != highScores.size(); ++i) {
262             if (!highScores.get(i).isEmpty()) {
263                 saveFileWriter.println(highScores.get(i));
264             }
265         }
266         saveFileWriter.close();
267     } catch (FileNotFoundException e) {
268         System.err.println("Unable to open " + fname +
269             " for output; high score file not saved");
270     }
271 }
272 // ----- NESTED PRIVATE CLASSES -----
273

```

Listing 12.4: HighScoreLevel Loading and Saving

These two methods should look familiar: `loadHighScores` uses an eof-controlled loop to read `HighScore` records (format actually defined inside of `HighScore` so it is not of interest to us right now) and `saveHighScores` uses a count-controlled loop to go through the list of high scores writing them out to the output file (again, the format of the writing is determined by the `toString` method of `HighScore` so inside of that class it is necessary to make sure the formats match; out here in the level, we just let `HighScore` do the work).

We will look back at the level constructor which calls `loadHighScores`:

```

77     this.currentScore = currentScore;
78     highScores = loadHighScores(fname);
79     while (highScores.size() < MAX_HIGHSORE_COUNT) {
80         highScores.add(new HighScore(0, ""));
81     }
82     Collections.sort(highScores, Collections.reverseOrder());
83 }
84
85 /**

```

Listing 12.5: HighScoreLevel Constructor

The level is constructed with a current score, the score the player made in the last game of `BlockDrop`, and the name of the high score file. `loadHighScores` is used in line 80 to load the high scores from the file. If there

were fewer high scores than expected, then `highScores` is filled with unnamed 0 scores. This means that the list will never be empty so we don't have to check for that.

Finally, line 84 is executed. The `Collections` class, in the `java.util` package, is a class with a number of **static** methods that provide useful algorithms for working with collections. One such algorithm is sorting. `Collections.sort` comes in two flavors. The first takes just a `List` of `Comparable` objects. Note that `List` is an **interface** and an `ArrayList` is-a `List`. When provided with just a `List`, the sort is done in ascending order according to `compareTo`; the list has to be of `Comparable` so that they all have `compareTo`.

The second overload of `Collections.sort` takes a `List` of `Comparable` as before and a `Comparator`. We're not going to go into what a `Comparator` is except to say that it modifies the results of calling `compareTo`. In the case of the `Collections.reverseOrder()` `Comparator`, it just reverses the sign of the result, thereby reversing the sorting order. Thus the list of high scores will be sorted but from highest down to lowest.

```

181     Collections.sort(highScores, Collections.reverseOrder());
182     saveHighScores();
183     if (showScores != null) {
184         removeSprite(showScores);
185     }
186     showScores = makeShowScoresSprite();
187     addSprite(showScores);
188 }
189
190 /**

```

Listing 12.6: `HighScoreLevel.insertScore`

The other place we need to sort the list is when we add a new high score. In Listing 12.6, you can see that the last score in the list is set to a new `HighScore` object with the new score and name. Then `highScore` is sorted (in reverse order). Then the high scores are saved and the screen is fixed up with a new `showScores` sprite. We will examine when this code is called in the last section of this chapter when we describe how the `BlockDrop` and `HighScoreLevel` games communicate.

Java packages

When do we break a method down into several smaller methods? One answer is when we are doing stepwise refinement and working our way down from the main problem. We define a solution in terms of simpler problems which we pretend are working, putting off defining the additional methods until we think about the next lower layer of abstraction.

There is one other time: when we find a method growing too long or taking care of too much. Think back to Chapter 4 and the `EasyDice` program (see Listings 4.12-4.14). The game had two states: rolling the dice or waiting for the user to place a bet. Each one did something different when `advance` was called. When the author first wrote the program, he put all of the processing for both states directly in the `advance` method. The method was over 40 lines long and handled checking what state the game was in and then, in both the `if` and `else` branches it checked if the button was pressed, and then it... Needless to say, breaking the method into three pieces, `advance` to determine game state, `advanceWaiting` to handle advancing one frame in the waiting state, and `advanceRolling` to handle advancing one frame in the rolling state, greatly improved the readability of the code. It limited the amount of context necessary to read any one of the three methods.

A similar problem occurs when writing a program with many classes. `BlockDrop` has fifteen different classes. When looking at a listing of all of the class names, it can be overwhelming. The context a programmer must comb through to find one desired class is too much. Java supports limiting the context by grouping classes together into **packages**. With good package naming it can be simpler to find a given class and, by keeping related classes together, it can be easier to reuse code in other projects.

The `BlockDrop` project has fifteen different class files. More than half of them are `Piece` and its subclasses. It would be nice if those files could be segregated off by them selves. Then there is `HighScoreLevel` and `BlockDrop`: they are separate levels and should probably be separated in some way (they don't belong in the same context). In fact, the files in this project can be divided into four different groups:

```

core                                + classes for main game level
  BlockDrop.java                    - main Game
  Board.java                        - where the pieces fall
  NextPieceBox.java                 - generates and shows new pieces
  ScoreSprite.java                  - show the current score
highscore                            + classes for high score level
  HighScoreLevel.java               - Game handling high scores
  ScrollingMessageBox.java          - looping, scrolling string table
pieces                                + piece classes
  I_Piece.java                      -
  J_Piece.java                      -
  L_Piece.java                      -
  O_Piece.java                      -
  Piece.java                        - superclass of all pieces
  S_Piece.java                      -
  T_Piece.java                      -
  Z_Piece.java                      -
util                                  + general utility classes (core & piece)
  Position.java                     - a position on the Board: row, column

```

Our source files are stored in a directory named for the current chapter, Chapter12. Splitting the files into the four named packages requires four steps:

- Move the source files to package subdirectories.
- Add a **package** line to each file.
- Add **import** for all classes not in the same **package**.
- Change the command-line for compiling and running the program.

A *package subdirectory* is just a subdirectory with the same name as the **package** it contains. Again, Java simplifies its job of searching for needed code by making the file system and Java name the same (just like class names and .java file names). Using our operating system we create the four subdirectories and move the source files down into them. The exact commands to create the directories depend on the operating system. We will just take a look at the output of listing the contents of the Chapter12 directory and its subdirectories. `ls *` lists the directories and their contents:

```

~/Chapter12% ls *
core:
BlockDrop.java Board.java NextPieceBox.java ScoreSprite.java

highscore:
HighScoreLevel.java ScrollingMessageBox.java

pieces:
I_Piece.java L_Piece.java Piece.java T_Piece.java
J_Piece.java O_Piece.java S_Piece.java Z_Piece.java

util:
Position.java

```

The **package** line is the key word **package** followed by the name of the package (the directory name) and a semicolon. The **package** line must be the first (non-comment) line in the file. It must be before any **import** or **class** or **interface** lines. Thus the first line of `HighScoreLevel.java` is:

```
1 package highscore;
```

Listing 12.7: HighScoreLevel package

The Java template for the **package** line is:

```
package <packageName>;
```

When the HighScoreLevel is showing, the player is expected to press the space bar to start the game (or, alternatively, to start at a higher level, press a digit for the starting level). When the key is pressed, the HighScoreLevel creates a new BlockDrop game with the appropriate level and then finishes:

```
159     if (keyPressed()) {
160         char key = getKeyPressed();
161         if ((key == ' ') || (('1' <= key) && (key <= '9'))) {
162             int level = 1;
163             if (('2' <= key) && (key <= '9')) {
164                 level = key - '0';
165             }
166             addGame(new BlockDrop(level));
167             finishGame();
168         }
169     }
170 }
171
172 /**
```

Listing 12.8: HighScoreLevel advanceScrollingScores

There is a problem for Java now. BlockDrop.java is not in the current directory when HighScoreLevel.java is compiled. You never had to think about **import** for any classes you wrote because they were always in the same directory as every other class you wrote (at least for any one project). Java just looked for the matching .java file in the current directory and all was well. Now, with multiple packages, you must explicitly tell Java where to find any classes not in the same package as the current class.

Given that the BlockDrop class is in the core package, HighScoreLevel needs the following include (so that line 168 can compile):

```
14 import core.BlockDrop;
```

Listing 12.9: HighScoreLevel import BlockDrop

(One other thing to note in advanceScrollingScores: we take advantage of the fact that the digit character codes are contiguous. In line 166 we convert from the code of a digit, say '4', to the number 4 by subtracting the code for '0' from the value of key. If we pretend that the digits begin at code 150, then, because they are contiguous, '0' = 150, '1' = 151, '2' = 152, etc. Thus '4' = 154. '4' - '0' = 154 - 150 = 4. This calculation does not depend on the particular value for the code of '0'; in fact, 150 is *not* the code of '0' in ASCII code. What value it is has is not important. The contiguous nature of digits (and lowercase letters, and uppercase letters) is the important thing to remember.)

We run javac and java in the *chapter root* directory. That used to be the same directory where the source code was; now the source code is below the root directory. We need to tell javac and java how to find the files.

```
~/Chapter12% javac -classpath ./usr/lib/jvm/fang.jar core/BlockDrop.java
```

Notice here that the name of the *file* to compile is the name of the *directory* followed by a slash (some operating systems use a different character) and then the full name of the file. The compiler is run in the directory above any package directory.

```
~/Chapter12% java -classpath ./usr/lib/jvm/fang.jar core.BlockDrop
```

Notice in the java execution line that the name of the *class* to run is the name of the *package* followed by a dot followed by the name of the class. We never noticed that javac works with file names and java works with class names before (except for leaving off . java) before.

Moving files to packages lowers the cognitive overhead in keeping track of context when looking at them. Grouping files in well-named packages *documents* what they have in common and guides the reader of the code to find what they need. Using packages requires moving the files, adding package and import lines, and changing the way you compile and run the program. The payoff, in ease of programming, is well worth a little bit of effort.

This section introduced some basic concepts of software engineering: Java **inter**faces, nested classes, the use of built-in algorithms (Collections is your friend), and using **packages** to group your classes and make reading your code simpler. All of these were presented in terms of making code easier to understand, one of the driving forces in writing code for this book. When you go on to study software engineering in more depth you will find that most of them have even deeper benefits to your code and the programmers who write and use it.

12.3 When It's Safe to Move: Collision Detection

is Tetris, or rather BlockDrop played? That is, how does a piece travel down the screen? In the original game, a piece appears, approximately centered, in the top row of the screen. At discrete time intervals it moves down the screen one block height (where each tetramino is composed of four blocks) at a time until it comes time to move and it cannot move down. Then the blocks freeze on the screen in their current position.

We will create each Piece as a FANG CompositeSprite. That way we can put the blocks into the Piece simply by setting their positions in the internal coordinate system.

Moving a Piece a discrete amount is actually quite simple. Rather than calling translate every frame with the product of the velocity and the frame time, we will use a countdown timer to wait for some number of seconds and then translate the Piece by a full block size. This does mean that the block size (the apparent size of the blocks in the Board's coordinate system needs to be known but that can be calculated when scaling the Piece. Actually, if all the Piece classes were the same size, then it would be easy to calculate. We will come back to this *same size* idea below.

The idea of a countdown timer was used in Chapter 5 (see Listings 5.2 and 5.1). We set a timer to the interval between moves, decrement the timer value by dT each time through advance, and when it crosses 0 the timer has expired.

```

76
77     moveClock -= dT;
78     if (moveClock <= 0.0) {
79         updateBoard();
80         moveClock += moveDelay;
81     }
82
83     checkKeyPressed();
84
85     int newScore = score.getScore();
86     updateLevel(oldScore, newScore);
87 }
88
89 /**

```

Listing 12.10: BlockDrop advance

We move the Piece and reset the timer and do it all over again as shown in Listing 12.10. Each time moveClock expires, updateBoard is called (which moves the block down) and the timer is reset. The use of += is to keep the time between movements as regular as possible; if one movement runs a little over, the next movement comes a little bit earlier.

```

182     return;
183     }
184     board.add(nextPiece.getPiece());
185     nextPiece.nextPiece();
186     }
187     }
188
189     /**
190     * Update the displayed level on the screen. The level is the number
191     * of times {@link #LINES_PER_LEVEL} rows have been cleared since the

```

Listing 12.11: BlockDrop updateBoard

The `updateBoard` method uses the `moveDown` method defined in `Board` to determine whether the current piece moved down or it was frozen. If it was frozen, then the game might be over. If not, move the `Piece` from the `NextPieceBox` (called `nextPiece`) and have it generate a new, random `Piece`.

How can `moveDown` determine whether or not it is safe to move the current piece down one row? And what, exactly, is a row and a column? What are the implementation details of `Piece` and `Board` and how do they work together?

A Square Grid

Any *Tetris*-clone has to deal with making sure that pieces are only moved or rotated when doing so does not move the piece (or some of its blocks) into an illegal position. An illegal configuration would have two blocks super-imposed on the board or one or more block off of the board. This is an instance of *collision detection*: would moving the piece to the proposed new position collide with any of the frozen blocks on the board (or the edge).

The collision detection needs to be done *before* moving the piece (or, alternatively, the piece has to be able to “back up” to where it was before the move) so that the piece never ends up in an illegal position. This means that our standard FANG intersects reaction after collision will not work.

How is a `Piece` stored? To make scaling pieces simple (all blocks in all pieces must have the same visual dimension *and* must be the same size as frozen blocks), all `Piece` objects should be square. Javier López, in his on-line *Tetris*-cloning tutorial[L608], simplifies things still further by placing all tetraminoes inside of a 5x5 grid.

Figure 12.3 shows three different pieces as they appear inside the grid in each of their four rotations. Using a 5x5 grid makes sure that all rotations of every piece fit completely within the grid; this is one of López’s most effective simplifying contributions. The numbers along the bottom of the figure are the facing value (facing is a field of the `Piece` class). Rather than being able to rotate freely, the `Piece` is constrained to be in one of four facings. We number them [0-3] so that we can use modular arithmetic to cycle around the different facings.

The darker square marks the *pivot* square. It is the center of the grid. The piece rotates around that square. It is also the center of the `CompositeSprite`, the point around which it will rotate. This makes showing the `Piece` in different orientations very easy.

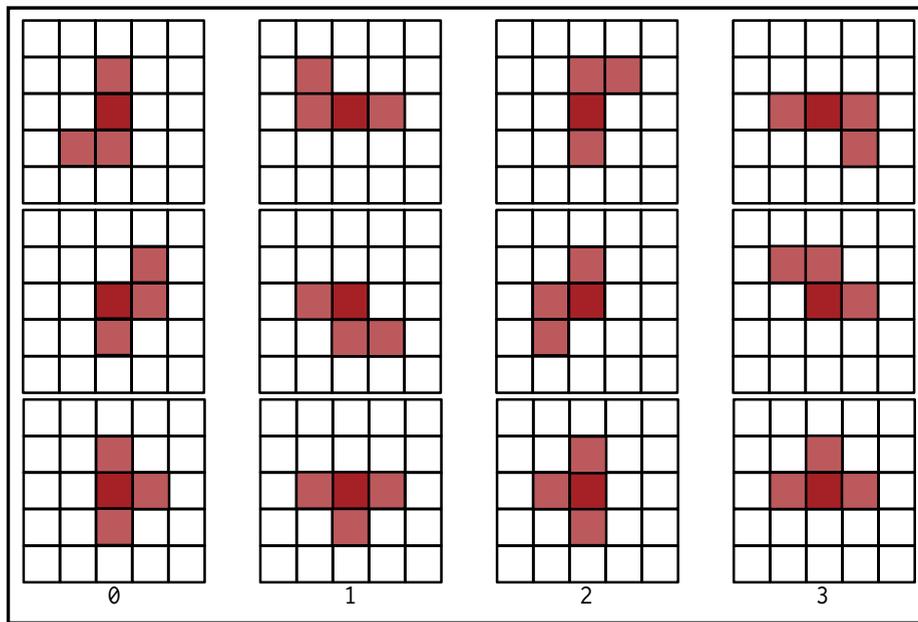
There are two different representations for the same thing: the blocks which make up a `Piece` are represented by actual `RectangleSprite` objects with given positions inside the coordinate system of the sprite and also inside a 5x5 grid of references to `RectangleSprites`. Any given location in the grid is `null` if that position is empty in the piece and refers to one of the blocks making up the `Piece` otherwise.

This same dual view is used in the `Board` class. The `Board` is a `CompositeSprite` but it also maintains a grid of blocks where `null` means the position is free and non-`null` means that the position is occupied by the given `RectangleSprite`. The number of rows and columns in the `Board` grid is set when the constructor is called.

```

62
63     /** height, in blocks, of the board */
64
65     this.rows = rows;
66
67     this.columns = columns;
68

```

Figure 12.3: *BlockDrop* Tetraminoes Grid

```

89
90     blockSize = 1.0 / rows;
91
92     // Show one square width on each of 3 sides
93     edges = new RectangleSprite((columns + 2) * blockSize,
94         (rows + 1) * blockSize);
95     edges.setLocation(0.0, blockSize / 2);
96     setEdgeColor(DEFAULT_EDGE_COLOR);
97     addSprite(edges);
98
99     background = new RectangleSprite(blockSize * columns, 1.0);
100    setColor(DEFAULT_BACKGROUND_COLOR);
101    addSprite(background);
102
103    ulcX = -blockSize / 2 * (columns - 1);
104    ulcY = -0.5 + (blockSize / 2);
105
106    blocks = initBlocks();
107 }
108
109 /**

```

Listing 12.12: Board Constructor

Both `Board` and `Pieces` have `ArrayLists` named `blocks` (or `blocksByFacing`; explanation below). These are the grids stored as lists of rows, each row being a list of `RectangleSprites`. The `initBlocks` method creates a 2-dimensional list of lists of blocks, initializing all values to `null`. In the `Board`, this method is called once and the grid is then used to play the game. When a `Piece` is constructed, it is called four times, once for each facing the piece can have.

```

31  * description of the piece in that facing. Blocks are either null
32  * (empty) or non-null (full), referring to a {@link RectangleSprite}
33  * that goes in that {@link Position}.
34  */
35  private final ArrayList<ArrayList<ArrayList<RectangleSprite>>> blocksByFacing;
36
37  /** The number of rows (columns) in this {@link Piece}'s grid */
71  this.columns = 5;
72  setColor(color);
73  position = new Position(0, 0);
74  this.facing = facing % 4; // make sure its safe
75  setRotationDegrees(90 * this.facing);
76  blocksByFacing =
77      new ArrayList<ArrayList<ArrayList<RectangleSprite>>>();
78  // for each facing, insert an empty blocks grid
79  for (int f = 0; f != 4; ++f) {
80      blocksByFacing.add(initBlocks());
81  }
82  }
83  }

```

Listing 12.13: Piece Constructor

The `blocksByFacing` is, conceptually, four copies of the `blocks` field in the board. Just as Figure 12.3 shows four different facings for each of the pieces in it, so, too, does each `Piece` have a list of block grids, one for each facing. Rather than create each `RectangleSprite` in each of the facings, the four blocks grids each refer to the same four blocks. We will defer talking about that until after discussing collision detection. For now just accept that the drawings in the figure represent the facing 0-3 blocks grids. Empty squares are `null` and filled squares are references to `RectangleSprites`. Thus there are 25 references and 4 are to actual sprites.

How can we keep track of a `Piece` on the Board as it falls? It is enough to know the *board* position of some reference square in *piece*. Because we use position a lot, it makes a useful class. A `Position` has a row and a column field, both `int` and both with a getter and a setter. This lets us track the `Position` of a `Piece` by having a field, `position`:

```

43
44  /** Current position of this {@link Piece} (in unit-squares) */
45  private final Position position;
46
47  /**

```

Listing 12.14: Piece Position

Along with the position, we also keep track of the current facing. Each facing is a quarter of a turn off of the previous facing. This means that there are at most four different facings since four quarter revolutions is one full revolution, returning the spinning object back to its original orientation. Thus we track facing as an `int` modulus 4.

The different facings have blocks in different locations. When a `Piece` is constructed, each different type of piece fills in a list of lists of `Positions`: it is a list with four lists in it, one for each *block* in the piece; each list has four `Positions` in it, one for each facing:

```

49  * facings so 4 different positions for each block. These are the only
50  * non-empty locations in the cells for each facing. Note that the
51  * every block appears, by reference, in each facing of cells.
52  * Protected so that subclasses can set the value after calling the
53  * {@link Piece} constructor.

```

```

54  */
55  protected ArrayList<ArrayList<Position>> blockPositionByFacing;
56
57  /**

```

Listing 12.15: Piece blockPositionByFacing

The blocks in a piece can be numbered, 0-3 (there are 4 blocks in every tetramino). Each block has a position inside the grid for each facing. Figure 12.4 repeats the figure of the blocks within a grid but this time each block in facing 0 is given a number.

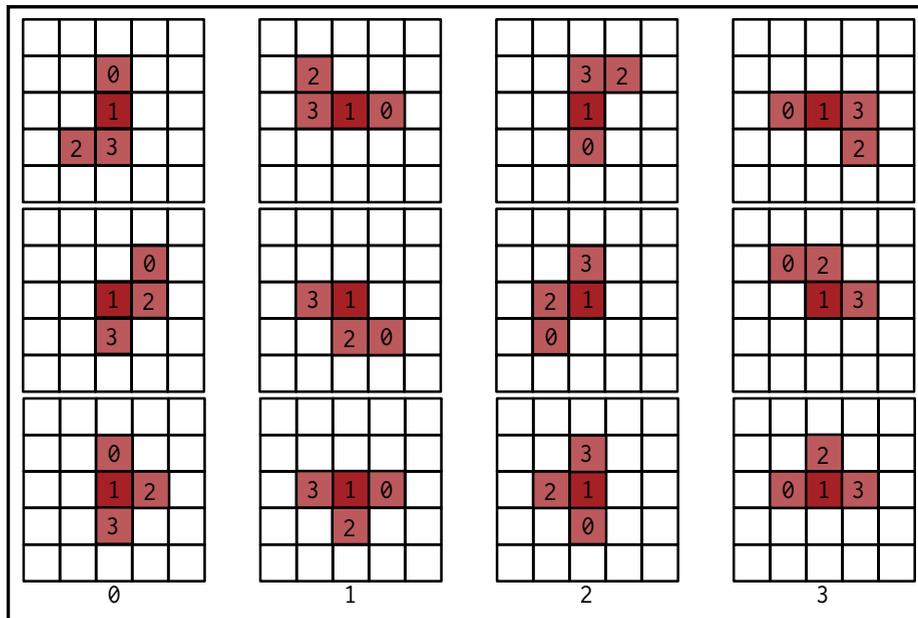


Figure 12.4: BlockDrop Tetramino Blocks

The numbers are assigned, in facing 0, from the top row to the bottom row and, in any given row, from the left to the right. Then, as the piece is rotated in each facing, each number follows the block to which it was assigned. That is, in the J piece in the top row, block 0 begins in position row 1, column 2 or (1, 2). When the piece is rotated one quarter turn clockwise, the block moves to position (2, 3) in facing 1. Then, subsequently, it moves to (3, 2) and (2, 1) in the next two facings.

The list of Position objects with values (1, 2), (2, 3), (3, 2), (2, 1) is the value of entry 0⁹ of blockPositionByFacing in J_Piece:

```

17  blockPositionByFacing =
18  new ArrayList<ArrayList<Position>>(Arrays.asList(
19  new ArrayList<Position>(
20  Arrays.asList(new Position(1, 2), new Position(2, 3),
21  new Position(3, 2), new Position(2, 1))),
22  new ArrayList<Position>(
23  Arrays.asList(new Position(2, 2), new Position(2, 2),
24  new Position(2, 2), new Position(2, 2))),
25  new ArrayList<Position>(
26  Arrays.asList(new Position(3, 2), new Position(2, 1),

```

⁹Tracking block 0.

```

27         new Position(1, 2), new Position(2, 3))),
28     new ArrayList<Position>(
29         Arrays.asList(new Position(3, 1), new Position(1, 1),
30             new Position(1, 3), new Position(3, 3))));
31
32     initialOffsetByFacing = new ArrayList<Position>(Arrays.asList(
33         new Position(-1, -1), new Position(-1, -1),
34         new Position(-1, -2), new Position(-2, -1)));
35
36     generateBlocks();
37 }
38 }

```

Listing 12.16: J_Piece Constructor

Lines 22-23 contain the value for block 0's positions in the four facings. Lines 25-26, 28-29, and 31-32 contain the positions for each of the other labeled blocks in the J piece.

The code, lines 20-32, is a more complex application of the `Arrays.asList` method first presented in Section 10.3. The method takes an arbitrary number of objects and returns something implementing the `List` interface in the `java.util` package¹⁰.

One use of `Arrays.asList` is to provide literal values for an `ArrayList` (or any other kind of list); the `ArrayList` constructor can take a `List` and copy its contents into the new `ArrayList`.

Reading a complex structure like this, it is good to understand the type of the variable to which it is being assigned and then read from the inside out. This is an `ArrayList<ArrayList<Position>>`. Lines 21-23 are as far in as we can go. Line 21 calls `new` on an `ArrayList<Position>`. The `List` passed into the constructor is a list of four `Position` objects. Line 23 ends with a comma because the whole `ArrayList<Position>` is just one object being bundled into a `List` by the call to `Arrays.asList` on line 20. The four `new ArrayList<Position>` lines (21, 24, 27, and 30) are where the four lists that will be elements in `blockPositionByFacing` and they will be inserted in the order they appear in the source.

Thus `blockPositionByFacing.get(2)` traces the location of the block labeled 2. Notice that block 1 does not move (as you would expect from Figure 12.4).

```

381     }
382 }
383
384 /**
385  * Get the blocks grid corresponding to the current facing of the
386  * {@link Piece}.
387  *
388  * @return the blocks grid (2-d list of list of {@link
389  *         RectangleSprite} references)
390  */
391 private ArrayList<ArrayList<RectangleSprite>> getBlocks() {
392     return getBlocks(this.facing);
393 }
394
395 /**
396  * Get the blocks grid corresponding to a given facing.
397  *
398  * @return the blocks grid (2-d list of list of {@link
399  *         RectangleSprite} references)

```

¹⁰A `package` can be defined inside of another `package`. Thus there is a `java` `package` containing a `util` `package`. Objects defined in that `package` have `package java.util` as their `package` line. Further, there is, somewhere in the code, a directory called `java` which contains a directory called `util`.

```

400     */
401     private ArrayList<ArrayList<RectangleSprite>> getBlocks(int facing) {
402         return blocksByFacing.get(facing);
403     }
404
405     /**
406     * Set the facing of the {@link Piece}; protected because it is not
407     * part of the interface, just the {@link #rotateCW()} and {@link
408     * #rotateCCW()} methods are exposed.
409     *
410     * @param facing the new facing value; well be reduced modulus 4.
411     */
412     protected void setFacing(int facing) {
413         this.facing = facing % 4;
414     }
415 }

```

Listing 12.17: Piece generateBlocks

The last line in `J_Piece` is a call to `generateBlocks`. The method is long because generating one block requires setting up both models of the `Piece`, both the grid which is described above *and* the `CompositeSprite` which displays the `Piece` on the `Board`.

The outer `for` loop loops over all of the blocks that make up the piece; the number of blocks is the number of entries in the `blockPositionByFacing` `ArrayList`. For all tetraminoes, this will be 4.

The block will be located (inside the `CompositeSprite`) in its facing 0 location. That way we can use the regular `rotateDegrees` method to rotate the appearance when changing the facing. Lines 390-391 get the facing 0 row and column for our block. Lines 394-395 calculate the location corresponding to the position.

Remember that the center of a `CompositeSprite` is (0.0, 0.0). Each block, in internal screens, is 0.20 wide/high. This is so the whole grid is 1.0 screens in each dimension. That makes the scaling on the board work well.

Lines 398-408 create the new block and add it to the `CompositeSprite`. The color of the block is the color of the `Piece` unless the block is in the pivot position. The pivot block is colored darker. This adds some interest to the colors of the pieces and makes it easier to see how rotation is working.

The final loop, lines 411-419, processes each of the four facings. For each facing the row and column where block belongs in the blocks grid is determined. Then the blocks grid, `facingBlocks`, is retrieved from `blocksByFacing`, the `blocksRow` is retrieved from `facingBlocks`, and, finally, the column position in `blocksRow` is set to refer to block. This last loop takes the blank grids built in the `Piece` constructor and populates them with the displayed blocks. Block 0 is put in at the appropriate positions according to Figure 12.4 (or rather, according to `blockPositionByFacing.get(0)`).

When To Move A Piece

With two grids, one modeling the `Board` and the other modeling the `Piece`, how can we determine whether or not it is safe to move the piece down? This is necessary because `advance` calls `moveDown` when the move delay timer expires and the method returns `true` if the piece was moved down one row and `false` if it could not be moved down and was, instead, frozen onto the board.

Before we look directly at when to freeze a piece, let's consider just how a piece moves. The previous section explained that we keep track of the position and facing of a `Piece` as it moves down the board.

Figure 12.5 shows a J-piece at several points in its journey down from the top of the screen. Figure 12.5.a shows the piece with a facing of 0 positioned at the top of the board. The position of the piece is (-1, 3) (shown at the top of the figure). That is the position of the upper-left grid square in the `Piece` grid (the one marked with the X) in the grid coordinates of the `Board`.

That last bit is important: the position of the `Piece` is expressed in the grid coordinates of the `Board`. Since we are tracking the upper-left corner of the `Piece` grid, the position of that grid square is always (0, 0) *inside* the `Piece`. It is important to keep in mind the difference of the two coordinate systems.

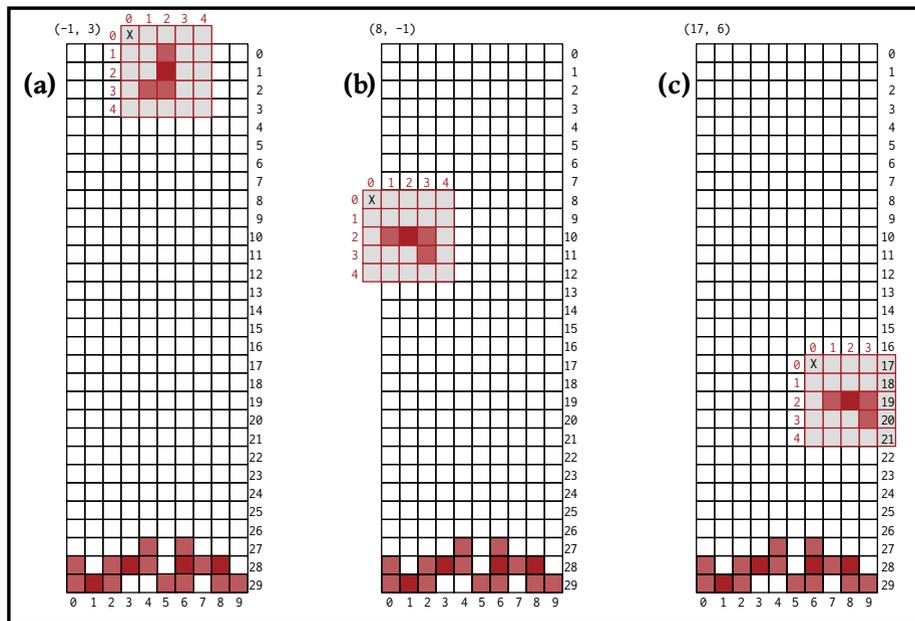


Figure 12.5: Falling Piece

Figure 12.5.b shows the same J-piece later in its descent. It has been rotated to facing 3 (according to Figure 12.3) and moved to the left edge of the Board. The Piece is positioned at (8, -1). The piece cannot be moved further to the left *in its current facing* because some of its blocks (to be differentiated from its grid squares) will be off of the board. The limitation to a column coordinate of -1 depends on the facing of 3. In facing 2, there are two blank columns on the left edge of the piece grid so it could move further over in that facing.

The core collision detection method is in Board. It is called `canMoveTo` and it takes a Piece and a board position (row, column, and facing). The board position is provided in parameters because that way we can test a piece in different positions before we decide to actually move it.

```

31  /** the default color of the main board */
32  public static final Color DEFAULT_BACKGROUND_COLOR = Game.getColor(
33      "misty rose");
34  /** the default color of the border around the board */
35  public static final Color DEFAULT_EDGE_COLOR = Game.getColor(
36      "dark slate blue");
37  public static final int HIGH = 2;
142  int canMove = CLEAR;
143  for (int r = 0; (canMove == CLEAR) && (r != piece.getRowSize());
144      ++r) {
145      int boardRow = row + r;
146      for (int c = 0;
147          (canMove == CLEAR) && (c != piece.getColumnSize()); ++c) {
148          int boardCol = column + c;
149          if (piece.hasBlock(r, c, facing)) {
150              canMove = positionOnBoard(boardRow, boardCol);
151              if (canMove == CLEAR) {
152                  if (hasBlock(boardRow, boardCol)) {
153                      canMove = HIT;
154                  }

```

```

155         }
156     }
157 }
158 }
159 return canMove;
160 }
161

```

Listing 12.18: Board canMoveTo

canMoveTo returns an `int`; the list of constants in the listing (from the beginning of the `.java` file), lists the values it might return. `CLEAR` means that there were no problems: all piece blocks on the board, no piece blocks overlapping board blocks. The other 5 values indicate what kind of problem there was. `LOW`, for example, means that at least one piece block is past the bottom of the board.

Back to `canMoveTo`. It goes through each row in the piece (the `for` loop on lines 144-159). Note that the loop is both count-controlled and sentinel-controlled. The loop will exit as soon as a block in piece is found to not be `CLEAR`. The inner loop (lines 148-158) loops through each column in the piece. Inside the loop a check is made whether there is a block at position (r, c) . If not, then the empty grid cannot effect whether or not the Piece can go in the given position so we jump over the rest of the body of the inner loop.

If position (r, c) has a block, then we need to check if the corresponding board position, $(boardRow, boardCol)$ is on the board. `positionOnBoard` simply checks whether the $(row, column)$ position it is passed is on the board, or in the range $[0-rows)$ for `row` and $[0-columns)$ for `column`. Depending on which limit is violated, the method returns an appropriate non-`CLEAR` value¹¹.

If the J-piece in Figure 12.5.b were tested in the same facing and one column to the left, then the block at $(2, 1)$ would be at board location $(10, -1)$: if the `board` row or `board` column of a `Piece` block is off the board, then `canMove` is set to something other than `CLEAR` by `positionOnBoard`, the `if` statement at line 153 is `false` so we go back to the top of the inner loop, then back to the top of the outer loop, and `canMoveTo` returns a non-`CLEAR` value.

If the board position of the block is on the board, it still might be on top of a block already in the game. Check that using the `hasBlock` method of the `Board`. One advantage of using the same model of the grid for both `Piece` and `Board` is the reuse of method names which do the same thing. Once you understand that `hasBlock` will return `true` if a position is occupied by a block (either in `Board` or in `Piece`), then it is easy to logically use the `hasBlock` method.

Figure 12.5.c shows the same piece further down, now on the right side of the board. Now the piece cannot move to the right because it will have two blocks off of the board. How do we actually move a piece down?

```

205     currentPiece.moveDown();
206     return true;
207 } else {
208     freeze(currentPiece);
209     int moveScore = deleteAllFilledRows();
210     if (score != null) {
211         score.increment(moveScore);
212     }
213     return false;
214 }
215 }
216
217 /**

```

Listing 12.19: Board moveDownIfPossible

The `moveDownIfPossible` method asks the piece if it can move down. `canMoveDown` is a wrapper method which just fills in the position and facing of the `Piece` and calls `canMoveTo`, returning the value it gets back:

¹¹The method is not listed here for brevity.

```

131
132  /**
133   * Can the piece move right without colliding?
134   *
277   rotateDegrees(-90);
278   }
279

```

Listing 12.20: Piece moveDown and canMoveDown

The `canMoveDown` method exists so that we don't mix the logic of making the current piece move (`Board.moveDownIfPossible`) with the arithmetic of figuring out the row number (`Piece.canMoveDown`). The `Piece` then uses its reference to the `Board` it is on to call `canMoveTo` with the right parameters.

If the piece *can* move down, `moveDownIfPossible` moves the piece, again by telling the piece to `moveDown`. It should be noted that `moveDown` does *no* error checking. If the piece is ordered down when `canMoveDown` returns **false**, the consistency of the game world is broken. It is safe to call in line 207 (because of the surrounding `if` statement). Then `moveDownIfPossible` returns **true** (the piece moved).

If the piece could not move down, then the method returns **false** but not until it freezes the piece, removes all full rows, and increments the score.

```

391   int boardRow = piece.getRow() + r;
392   for (int c = 0; c != piece.getColumnSize(); ++c) {
393       int boardCol = piece.getColumn() + c;
394       if ((piece.hasBlock(r, c) &&
395           (positionOnBoard(boardRow, boardCol) == CLEAR)) {
396           RectangleSprite block
397               = acquireBlock(piece.blockAt(r, c), boardRow, boardCol);
398           blocks.get(boardRow).set(boardCol, block);
399           addSprite(block);
400       }
401   }
402   }
403   removeSprite(piece); // remove the CompositeSprite
404   }
405
406  /**

```

Listing 12.21: Board freeze

The `freeze` method has a structure very similar to that found in `canMoveTo`. It has two nested loops that go over the grid of the piece passed in as a parameter. here it uses the current position and facing of the block rather than providing one as a parameter. For each location in the piece grid, if it has a block and the corresponding board position is actually on the board¹² then the `Board` steals the block from the `Piece`. The `RectangleSprite` representing the block is returned with the call `piece.blockAt`. The block is rescaled and relocated according to the board's size and the `boardRow` and `boardCol` where the block is going. Then it is inserted into `blocks` and added to the `Board`'s list of sprites to display.

Look at lines 380-383 again. The block is retrieved from the `Piece` and the reference to it is used (in `acquireBlock`) to recolor it and set its *location*. Then the grid model of the `Board` is updated with the addition of the block at the correct position. Finally, the visual model (the `CompositeSprite` model) of the `Board` is updated with a call to `addSprite`.

¹²The `move...IfPossible` and `rotate...IfPossible` methods constrain movement; if they are the only ways the current piece was moved, this check is superfluous. The check for a position being on the board is not expensive either in running time for the program, nor for programmers understanding why it is called, so it remains here.

Hopefully you are a little bit concerned right now. Earlier we had four different grids, one for each facing, referring to each block in the `Piece`. Now we have not only the `Piece` but also the `Board` referring to same `RectangleSprite`. Is this legal? Is it sound?

It is legal: recall that any number of references can refer to a single object so references from different underlying objects is perfectly legal as far as Java is concerned. It is also sound *in this case*: line 405 removes the `Piece` from the `Board` so there will be no attempt to draw the block inside of the `Piece` ever again. It would still be sound, even if the `Piece` remained on the `Board` except for the fact that we rescaled and relocated the block. Comment out line 405 and see what happens¹³.

So, we transfer ownership of the blocks from the `Piece` and then stop paying attention to the `Piece`. Out in `moveDownIfPossible`, after freezing, the board clears out any filled rows. A row is filled if all positions in the row have a block in them; alternatively stated, a row is filled if it has no `null` references.

```

449  /**
450   * Is the given row full of blocks?
451   * @param rowNdx the index into {@link #blocks} of the row to check
452   * @return true if row is filled with blocks; false otherwise
453   */
454  private boolean isRowFilled(int rowNdx) {
455      ArrayList<RectangleSprite> currRow = blocks.get(rowNdx);
456      int c = 0;
457      for (c = 0; c != columns; ++c) {
458          if (currRow.get(c) == null) {

```

Listing 12.22: Board `isRowFilled`

Given a `rowNdx`, `isRowFilled` traverses across the row using a count-controlled/sentinel-controlled loop. This time, for variety (and ease of understanding the purpose of the code) the sentinel appears *inside* the loop. The `if` statement on line 453 is the sentinel condition stated positively. The `break` statement on line 454 “breaks out” of the inner most loop construct in the current scope. Thus `break` stops execution of the `for` loop, continuing execution with the first line *after* the loop. This means that we can tell if the loop terminated by the count or by the sentinel by looking at the count-control variable. Since we want to check the value of `c` after leaving the loop, the variable must be declared *before* the loop. The row is filled if the `for` loop ended because `c == columns`.

To delete all filled rows, loop across all rows in the board and use `isRowFilled` to determine if it should be deleted.

```

345      for (int r = 0; r != rows; ++r) {
346          if (isRowFilled(r)) {
347              deleteRow(r);
348              ++rowsDeleted;
349          }
350      }
351      return rowsDeleted;
352  }
353
354  /**

```

Listing 12.23: Board `deleteAllFilledRows`

The check of rows has to go from top to bottom. If it went the other way, `deleteRow` on row `r` would move the unchecked row `r - 1` into the `r` row (all rows above `r` have to move down when the row is deleted; more just

¹³Little copies of the pieces begin to appear. They are scaled down because the `Board` has more blocks in its grid so each block is smaller. They are offset from the piece on the screen because the position depends on where the piece last was on the `Board`. Commenting out a line of code is one way of trying to figure out what it does; you have to be careful not to randomly change things, though. Try to predict what the result will be to check if you really understand the code.

below). Then the increment in line 347 would move the index above the unchecked line. Moving down this will not happen (the already checked line above the full line falls and then the loop moves to the next lower line, the next unchecked line).

```

361     for (int r = deadRowIdx; r != 0; --r) {// counts backwards!
362         blocks.set(r, blocks.get(r - 1));// (r - 1) >= 0
363         ArrayList<RectangleSprite> currRow = blocks.get(r);
364         for (int c = 0; c != columns; ++c) {
365             if (currRow.get(c) != null) {
366                 currRow.get(c).setLocation(locationFromRowColumn(r, c));
367             }
368         }
369     }
370     // add a new, empty row at the top.
371     blocks.set(0, initRow());
372 }
373
374 /**
375  * Remove the sprites in the deadRow from the display of the {@link Board}
376  * @param deadRow the row of sprites (and, perhaps, nulls) to remove
377  */
378 private void removeRowOfSprites(ArrayList<RectangleSprite> deadRow) {
379     for (int c = 0; c != columns; ++c)
380         if (deadRow.get(c) != null)
381             this.removeSprite(deadRow.get(c));
382 }
383
384 /**

```

Listing 12.24: Board deleteRow

In `deleteRow`, first all of the blocks in the dead row are removed from the `Board` (the `CompositeSprite` model) using `removeRowOfSprites`. While it is assumed that `deleteRow` is called only on filled rows, it is a good idea to check the blocks to make sure you don't try to remove `null` values. `removeRowOfSprites` shows how the curly braces after a `for` or `if` statement are optional: the `if` controls the execution of the next statement; if there is only one statement, the curly braces are optional and if there are more than one, the curly braces make the multiple statements into a block which can be treated as a single statement.

At line 363 the rows are taken from the dead row's index up to 0. This `for` loop counts backwards. Each time through the loop the row above (at index `r - 1`) is moved down in `blocks` (line 367). That fixes up the grid model. The blocks in the row that moved must be relocated, too. That is what the loop in lines 369-372 does. Finally, after row 0 was moved down to row 1, make a new row 0. `initRow` was used in building the new, empty board grid and returns a new, empty row, just what we need at index 0.

`deleteAllFilledRows` returns the number of rows which are deleted; the `Board` adds that value to the score.

This section examined using two different models for a game simultaneously. While it might seem contrary to the DRY principle, two different models are not repeating one another but serving different purposes. In `BlockDrop` the two models are the grid and the `CompositeSprite`. Blocks (`RectangleSprites`) are kept in both models.

The simple, grid-based model is used to check for collisions and to keep track of the blocks. By having all facings of a given `Piece` take up the same space, rotating a piece does not require any fixing up of the position.

The `CompositeSprite` model, the screen model used by `FANG` since the beginning of the book, uses `double` values to keep track of locations. Blocks must be scaled so they appear properly within the visual representation.

The next section looks at how another *factory* is used as a polymorphic constructor, how the high score list is implemented using a `CompositeSprite` with its own `advance` method, and a few other small finishing details for `BlockDrop`.

12.4 Finishing BlockDrop

The whole `BlockDrop` program, with all the pieces, the game, the board, and the high score handling, runs just about 35 pages. That means there is not enough room in this chapter to carefully examine every line. Earlier we looked at how a given `Piece`, the `J_Piece` is constructed. The other six pieces are the same except that the locations where the blocks go in the grid are specific to the given piece shape.

The `ScoreSprite` is simply a `StringSprite` with an `int` `score` field with a `get` and a `set` method. When the score is set, the text of the `StringSprite` is updated to reflect the new score. The implementation details are so similar to other sprites from earlier chapters that detailed study of the code is left as an exercise for the reader.

`BlockDrop` implements a simple level mechanism: every 10 filled rows, the speed of the game goes up by 5%. The level is advanced whenever the tens digit of the score changes as a result of freezing a piece. The code for doing this is called `advance` in `BlockDrop`; it will not be explored further here.

The remainder of this section will look at the `NextPieceBox`, a sprite (for showing the next piece coming up in the game) and a factory for creating random `Pieces` and a look at creating a random *contrasting* color, and finally a look at the `ScrollingMessageBox` in the `HighScoreLevel`.

Another Factory

Part of the gameplay of `BlockDrop` is being able to see into the future. While maneuvering a piece as it falls, the play can see the next piece which will be served up. In FANG terms, this means the next `Piece`, a `Sprite`, must be displayed on the screen. To keep the scale of the `Piece` consistent with the size it will have on the `Board`, the new `Piece` is generated by a `CompositeSprite`-derived class called the `NextPieceBox`. The `NextPieceBox` is a square scaled to match the grid of one `Piece`. The scaling is accomplished by passing the `blockSize` determined by the `Board` into the `NextPieceBox` constructor.

The piece inside the box can be accessed through a getter method; that is necessary so that when the current piece is frozen in the `Board`, the piece in the `NextPieceBox` can be added to the game. The one other public method of `NextPieceBox` is `nextPiece`. Calling that method has the box create a random new `Piece`. Since there are seven different kinds of pieces, the code to do this is a factory.

```

70 public void nextPiece() {
71     // remove old piece from display
72     if (piece != null) {
73         removeSprite(piece);
74     }
75
76     // randomize piece, facing, and color
77     int randomPiece = Game.getCurrentGame().randomInt(
78         Piece.PIECE_COUNT);
79     int facing = Game.getCurrentGame().randomInt(4);
80     Color randomColor = getRandomContrastingColor(getColor());
81
82     // the factory code; call the right constructor
83     if (randomPiece == 0) {
84         piece = new I_Piece(facing, randomColor);
85     } else if (randomPiece == 1) {
86         piece = new L_Piece(facing, randomColor);
87     } else if (randomPiece == 2) {
88         piece = new J_Piece(facing, randomColor);
89     } else if (randomPiece == 3) {
90         piece = new Z_Piece(facing, randomColor);
91     } else if (randomPiece == 4) {
92         piece = new S_Piece(facing, randomColor);
93     } else if (randomPiece == 5) {

```

```

94     piece = new O_Piece(facing, randomColor);
95 } else if (randomPiece == 6) {
96     piece = new T_Piece(facing, randomColor);
97 }
98     addSprite(piece);
99 }

```

Listing 12.25: NextPieceBox nextPiece

The `nextPiece` method is broken into three parts: get rid of the old piece (if there was one), generate random piece number, facing, and contrasting color, and finally call the right constructor to create the piece that was randomly selected.

Lines 71-75 are straightforward. The old `Piece` has probably been added to the `Board` so it should no longer be displayed.

Lines 76-80 are also understandable, so long as we accept that `getRandomContrastingColor` does what its name claims.

In `TwentyQuestions` the factory code used a `String` value found in a text file to determine what kind of `AdventurePage` to create (see Section 11.1). Here, instead of encoding the class of the `Piece` as a `String`, we encode it as an `int`. Since there are 7 possible pieces (the number is stored in the `Piece.PIECE_COUNT` constant), we generate a non-negative `int` less than 7 and then use a multi-way `if` to call one of 7 constructors. The result is stored in a reference to a `Piece` and the `Piece` is added to the `NextPieceBox`'s display.

Contrasting Colors

How can `getRandomContrastingColor` work? First, what does it mean for two colors to be contrasting? For our purposes here, all colors are either light or dark and opposites contrast. How can we tell a light from a dark color? Take the three *channels* of color information, red, green, and blue. Each is an `int` on the range [0-256). Average the three channels together. If the average is greater than or equal to half of the range (128) then the color is light; less than half and it is dark.

```

134 private Color getRandomContrastingColor(Color bgColor) {
135     int low = 128; // assume we want a light color
136     int high = 256;
137     int averageBrightness =
138         (bgColor.getRed() + bgColor.getGreen() + bgColor.getBlue()) / 3;
139
140     if (averageBrightness >= 128) { // bgColor is light
141         low = 0; // need dark color
142         high = 128; // range
143     }
144
145     Game curr = Game.getCurrentGame();
146     Color randomColor = Game.getColor(curr.randomInt(low, high),
147         curr.randomInt(low, high), curr.randomInt(low, high));
148     return randomColor;
149 }
150 }

```

Listing 12.26: NextPieceBox getRandomContrastingColor

If the parameter `color`, `bgColor`, is dark, we will generate numbers in the upper half of the range for each color channel of the random color and if `bgColor` is light, the lower half. Lines 135-136 assume we want a light color. If the average brightness of `bgColor` is in the light range, change the high and low limits for the random numbers. Then, finally, just generate a color from three random integers.

Scrolling List

When we show the high scores list, how can we make it interesting? Further, how can we show scores than would comfortably fit on the screen at one time? The answer is that we could scroll the names on the screen. That is, have a `CompositeSprite` positioned so that everything below a certain point is not visible on the screen and move items up the screen, wrapping them to the bottom of the list when they go off the top of the `CompositeSprite`.

Everything is stuff we have done before: the `StringSprites` in the list are sprites with velocity, moving a sprite until it hits some horizontal line is dealing with falling apples, even relocating the sprite as it crosses one horizontal line back at the beginning horizontal line is from `NewtonsApple`. Doing all of this with several sprites at the same time in a `ArrayList` extends what was done in previous games but we have moved groups of sprites before as in `RescueMission`.

The public interface (not to be confused with a Java **interface**) for `ScrollingMessageBox` is a little like what we would have for a game because it contains other sprites and advances them.

```
class ScrollingMessageBox
    extends CompositeSprite {
    ScrollingMessageBox(...
    ScrollingMessageBox(ArrayList<String> msg, double velocity, int gap)...

    public void add(String newMsg)...
    public void advance(double dt)...

    // get/set for Gap, LineHeight, and Velocity

    public void setColor(Color color)...
}
```

Hopefully it is clear that the `get/set` comment expands into six methods. Further, it is hopefully obvious how to write the six methods (or will be once you know the types of the values). `setColor` is necessary so that the color set to the sprite is propagated to the lines. The `ScrollingMessageBox` has no background or edges; those must be provided by the client code if they are required.

The primary `ScrollingMessageBox` constructor takes three parameters: the list of messages to scroll, the velocity to scroll the messages, and the gap between the last message and the repeat of the first message. The gap is expressed in blank lines between the last message and the starting over of the message list. It increases the length of the cycle as if there were gap blank messages at the end of the list. The default constructor passes in an empty list of `String` and default values set in the **public static final** constants of `ScrollingMessageBox`. The line height, in internal screen coordinates, is set to a default in the constructor. It can be changed with the appropriate `get/set` methods.

```
91 public void advance(double dt) {
92     scrollMessages(dt);
93     wrapIfNecessary();
94 }
189 private void scrollMessages(double dt) {
200 private void wrapIfNecessary() {
201     StringSprite top = messages.get(indexOfTopLineOnScreen);
202     if (top.getMinY() < TOP_EDGE) {
203         int indexOfBottomLineOnScreen = (indexOfTopLineOnScreen +
204             (messages.size() - 1)) % messages.size();
205         StringSprite bottom = messages.get(indexOfBottomLineOnScreen);
206         top.setY(bottom.getY() + lineHeight); // top just below bottom
207
208         if ((indexOfTopLineOnScreen == 0) && (gap > 0)) {
209             top.translateY(lineHeight * gap);
```

```

210     }
211
212     indexOfTopLineOnScreen = (indexOfTopLineOnScreen + 1) %
213         messages.size();
214     }
215 }

```

Listing 12.27: ScrollingMessageBox advance

The primary public method is `advance` which calls two implementation methods, one to move all the lines according to the velocity and one to wrap. The movement is done in `ScrollingMessageBox.scrollMessages` rather than inside a `StringSprite`-extending sprite because all of the sprites share the same velocity.

The `wrapIfNecessary` method gets the `StringSprite` with the highest y-coordinate and checks if it is passed the top edge of the “screen”. If it is, the `StringSprite` is relocated to one line height below the line with the lowest y-coordinate. The scrolling is only designed to work going up the screen. If the line moved had index 0, then the gap is opened up between the last and first lines.

When we checked whether an apple hit the ground or if swimmers hit the edge of the screen, we checked *all* of the sprites. How does `wrapIfNecessary` find the `StringSprite` with the highest y-coordinate? Or the one with the lowest y-coordinate for that matter.

One way it could work is to search the `ArrayList<StringSprite> messages` for the highest/lowest y-coordinate (just call `getY()` on each element). This sprite uses an alternative: it has a field called `indexOfTopLineOnScreen`. The field is initialized to 0 because lines are located from top to bottom of the sprite. Line 212 in `wrapIfNecessary` advances the field to the next highest line on the screen *after* the old top is wrapped.

Finding the other extreme, the lowest y-coordinate, just means finding the one before. Lines 203-204 do the arithmetic. The expression in those two lines should be read as `(indexOfTopLineOnScreen - 1) % messages.size()`

The problem with using that simple statement is that computer modular numbers are not the same as mathematical modular numbers. In mathematics, modulo 17 (just to pick a modulus), $(0 - 1) \% 17$ is 16; this makes sense because $(16 + 1) \% 17$ is 0 and mod 17 arithmetic limits the values to $[0-17]$.

In Java and most computer languages, modular arithmetic limits the values for mod 17 to $(-17-17)$. That is, -1 is a valid value and makes sense to be the value of the expression $(0 - 1) \% 17$. Unfortunately, -1 is not a good value to use as an index into an `ArrayList`. So, instead of subtracting one, in Java we add the modulus minus one. That is, we evaluate $(0 + (17 - 1)) \% 17$. This expression evaluates to 16 in Java and to 16 in mathematics.

Since the modulus for the indexes of the messages list is the size of the list, on line 204 the size minus one is added to the index and then the modulus of that sum is taken. This makes sure that the value of `indexOfBottomLineOnScreen` is the index right before (in a cycle modulus `messages.size()`) `indexOfTopLineOnScreen`.

12.5 Summary

Java Templates

Chapter Review Exercises

Review Exercise 12.1

Programming Problems

Programming Problem 12.1

String Processing: Interactive Fiction

This chapter brings together all of the topics presented so far while examining a *text adventure game*. A text adventure game is a game where the player explores a virtual world; the exact goal of the game depends on the contents. It is called a *text adventure game* because the world and everything in it is described in text and the user's interaction with the game is by typing commands.

Creating the game engine is only half of the effort in creating a text adventure game: every object in the world must be described in a data file. To support game authors, this chapter examines a more flexible data file formatting, what Jon Bentley calls a “self-describing data file”[Ben88].

Text adventure games require more sophisticated *string processing* than we have yet seen. User commands are short declarative sentences like “get ticket” or “talk Dean of Students” or “move north”. The last one can be shortened to “north” in most games. Taking apart a sentence like this involves *parsing* the string, breaking the whole thing up into syntactic units. Remember that syntax, when applied to programming languages, talks about structure. The same is true when writing a simple parser: the parser breaks the string into tokens, words and symbols defined as being significant. Then the sequence of tokens is processed to figure out the meaning, the semantics, of the user command.

Similar parsing tasks are necessary when processing a self-describing file. Rather than having each record in a text file have exactly the same ordering, each field will be identified by name. The value of a field might be just one line or multiple lines long so the syntax of the data file must indicate where the field values begin and end and the parser must be able to handle them. Finally, printing descriptions for the user requires *wrapping* the text so no lines are too long to display. Wrapping is frequently used in word processors and other text editors.

The scale of the text adventure game program lends itself to *incremental development*, the creation of increasingly complete and complex “versions” of the program. Incremental development permits the programmer to focus on one piece at a time and is one component in the currently popular development methodology known as *agile* development.

First a quick introduction to what a text adventure game is and a high-level design of the game engine. Then the project will be broken into phases for incremental development and we will look at human-readable and editable data file formats.

13.1 Back to the Future: Interactive Fiction

As mentioned above, *text adventure games* are games where the player interacts with a virtual world through text: the user's commands are entered as lines of text, typically imperative sentences, and the results are described to the user with text on the screen. This section will give a broad overview of how a text adventure

game works and discuss the design of a game *engine* (it will also define the difference between a game and a game engine).

Text Adventure Games

When discussing the data structure at the heart of the `TwentyQuestions` program in Section 11.1 the Choose-Your-Own-Adventure style books were introduced. A given page as a text description of what has happened and the reader, as protagonist of the fiction, selects a response to what happened by choosing the next page.

The narrative nature of a text adventure game has led to the coining of the term *interactive fiction*. The Interactive Fiction Archive¹ maintains a large collection of interactive fiction as well as game systems for multiple computing platforms. Limited to text input and output, interactive fiction is well-suited to less CPU powerful computing devices such as smart phones and MP3 players. They are also well-suited to our abilities to read and write from standard input and standard output.

The World

The world of a text adventure game is a collection of *locations* which are connected, one to another. In these locations there are *critters*, characters inhabiting the virtual world. Locations and critters can have *items* in their possession which can be picked up by, stolen by, or perhaps traded for by the player. The player is also a critter because they have an *inventory*, a collection of items.

This description is intentionally very general. Exactly *how* locations are connected one to another depends on the world being modeled. If the game world is on the surface of the Earth, it would make sense to discuss locations being laid out in a grid with locations laying to the “north”, “south”, “east”, and “west” of a given location; it should be possible that a location have no location in a given direction so that the location is at the edge of the map in that direction. If the game world represents the interior of a submarine, “fore”, “aft”, “port”, “starboard”, “up”, and “down” might all make sense. A space station, a system of deep caves, or a collection of airports might each have a connection scheme different than either above.

Critters and items differ primarily in how the player can interact with them. A critter is something to which the player can *speak* or with which the player can *fight* or *barter*. Note that any given game might have none, some, or all of these options. A critter might represent a talking garden gnome, a superhero in blue spandex, or a clanking robot. A critter might even be a vending machine *if* the interaction with the vending machine is more like that of interacting with other actors in the drama rather than in interacting with *things*.

Items can be *picked up* or *dropped*. Some might be able to be *eaten* or *drunken*, *read* or, perhaps, even *combined* to make some new item. It is possible that the connections from one location to another might require the player to possess some particular item (generically a *key*) to pass or require the player to have met some particular critter.

Note that the last several examples show that text adventure games on a computer go well beyond what is possible in a Choose-Your-Own-Adventure-Book. In the book, no matter how you turned to page 57 the choices presented to you are the same. So if you got there after the second page you read or after bouncing through more than a hundred pages of epic battles, whatever happens on and after page 57 is the same. The player critter has state which can be checked at any time. If the player must have visited the great elf before leaving the forest, this can be enforced. If killing the dragon precludes any happy ending, then lock the happy ending choices after the dragon dies. The key here is that the computerized game has *state* which can be used to permit choices at one point in the game to have consequences in another part.

What fields define a location, critter, or item? Each game item will have a *universally unique identifier* (UUID). A UUID is unique within the collection of all objects in a given game. This permits any object to refer to another by its UUID. This should sound familiar: the page number in `TwentyQuestion` served as a UUID for the `AdventurePages`. We could, again, use a serial number to identify each item; the UUID of an object would be implicit in its position in the data file, a record number counting from the beginning of the file.

Consider how difficult it was to write and read the short data files for `TwentyQuestions`. Even with just 8 entries, keeping track of the index number of each and having a given question refer to its yes and no children was not always easy. Explicit UUIDs, included in each object’s entry in the data file, will make it much easier

¹<http://ifarchive.org/>

to refer from one object to another. What data type should the UUIDs have? The choices seem to be `int` or `String`. The `int` would make it possible to use the UUID as an index into an `ArrayList` to make looking things up easier.

Hopefully you read that last sentence with some discomfort. It mixes different levels of abstraction in a very dangerous way. When designing a class to hold associated information, it is not appropriate to worry about how a collection full of the class will be structured. It is enough to assume that we can look up a game object in a collection of game objects and, based on the UUID, find the one we want.

So, one option for UUID remains `int`. Does `String` have any advantages over `int`? For the game author the answer is, “Yes!”

Imagine that an item has a field, call it `owner`. The owner of an item is the critter or location where that item currently resides. The data file will contain something that says the owner of the *Dagger of Ensidor* is either 1003452 or **GrummTheBarbarian**. Which would make the data file easier for the human being to read, write, and debug?

An item, a location, and a critter each have a UUID. They also each have a name (the *Dagger of Ensidor* or *Grimm the Barbarian*), and a description (the text displayed on the screen when the player enters the location or sees the critter or item).

A location will also have some number of links to other locations. A link is, at least, the UUID of the location at the other end of the link. It might also include the direction of the link (so the data file indicated whether the location is to the “north” or the “south” of the current location). Again, explicit naming in the data file will make it much easier for game authors; otherwise every location must have 4 (or 6 or 8) outgoing links in a specific, implicit, order which the author must remember.

A critter will also have a current location (so that the game knows when to describe it) and an inventory. An inventory is a collection of items. An item has an owner.

These lists are minimal in the sense that there is no indication of what impact an item has on the game nor any way the game designer can determine how the player interacts with a critter. Later in this chapter we will discuss how to implement conversation and/or combat in the system and additional fields for all of the game objects to support that gameplay.

Game and Game Engine Design

What is an Adventure Game?

What is a Game Engine?

Playing a Text Adventure Game

13.2 Iterative Development

Creating Units of Useful Work

* What is the smallest thing that will work * Agile sidebar * Textadventuregame sidebar

* Attribute/Value pairs

The Game Loop

GameObject Interface

Location Class

Critter Class

Item Class

Connected Locations

13.3 Reading The Data

This section takes a bottom-up approach to design. Instead of dividing a problem into smaller and smaller pieces, this section will build up how a complex record could be stored. First we review the requirements of a self-describing data file and then look at how to implement them.

A self-describing data file is a text file that describes what it contains. This is, in a data file sense, exactly how we try to write Java code. Choosing good names, using indentation to document inclusion, and writing comments that document the intention, these all can be applied to *data* files.

Why do we apply those techniques to Java programs? They make the code more *readable* and *maintainable*. Code will be read many more times than it will be written; it is worth the investment to make it as readable as possible. Maintenance of software is rewriting it to remove bugs or to accommodate any changed program requirements. Good programming style and documenting the programmer's intent make the code easier to follow and much easier to edit when it is necessary.

These same techniques can be applied to data files so that the data can be *read* and, more importantly, *written* by a human being using a text editor. This means that the programmer can develop a series of test files to test the file reading code. It means game designers can work on data files *before* the game engine is finished. One big win in separating data from the code is that data changes can be made quickly; this is only true if the data can be understood so the *appropriate* changes can be made quickly.

A self-describing data file should

- permit comments - ignored by the system
- be free format - ignore most spaces
- explicitly identify each object and each field
- be order independent - with field identifiers the fields can come in any order
- permit multi-line entries

Low Level Reading

Back in Section 9.5, the `File` and `Scanner` classes were introduced. That section also looked at how to read a text file word-by-word and how to read a text file line-by-line; it is notable that we have not read a file or keyboard input *character-by-character*.

One of the greatest contributions of the original Unix operating system was the abstraction of all files (and devices like the network card) as streams of bytes. This uniform abstraction means a program can be written to work with a stream of bytes without caring about the origin of the stream: the user could be typing it in, it could be the result of requesting a given URL from a Web server, or it could be a file stored on the user's USB flash drive. This is an example of separating concerns: the code which *opens* a source interacts with the real source and that code is part of the operating system; the code that *uses* the stream of bytes is insulated from the details.

Similarly, in Java, we have been working with a high-level abstraction of input. Rather than working directly with streams of bytes or even streams of characters² we have worked with the `Scanner` class. The `Scanner` reads a sequence of characters (insulated from the source of the characters) and groups the characters together into *tokens*. A token is a sequence of characters fitting some syntactic description. We will work out the syntactic description used by `Scanner` by default and how to change it below.

²There is a distinction between the two: remember Unicode? Some characters take more than one byte to express.

Readers and Writers

Can we gain access to a stream of bytes like `Scanner`? If we can, what can we do with a stream of bytes which we could not do with the `Scanner`? Yes, we can, using the built-in family of classes known as Reader classes³.

There are a large number of Reader classes; they can be recognized by Reader ending their class names. We will look at two: a `FileReader` which can be constructed from a `File`, just like we have been constructing `Scanner`; and a `FilterReader`, a reader which is constructed as a wrapper around another Reader possibly modifying the stream of characters before passing the characters on.

We will start by writing a program, `CountCharacters` modeled on our earlier `EchoFile` program, which will read a file named on the command-line *character-by-character*, counting the total number of characters in the file.

```

35     * characters.
36     */
37     public void count() {
38         if (file.exists() && file.canRead()) {
39             FileReader reader = null;
40             try {
41                 reader = new FileReader(file);
42                 int characterCount = 0;
43
44                 int ch = reader.read();
45                 while (ch != EOF) {
46                     ++characterCount;
47                     ch = reader.read();
48                 }
49                 System.out.println(file.getName() + ": " + characterCount);
50             } catch (FileNotFoundException e) {
51                 System.err.println("PANIC: This should never happen!");
52                 e.printStackTrace();
53             } catch (IOException e) { // file was opened before this exception
54                 System.err.println("Problem while reading \"" + file.getName() +
55                     "\".");
56                 e.printStackTrace();
57             } finally {
58                 try {
59                     if (reader != null)
60                         reader.close();
61                 } catch (IOException e1) {
62                     System.err.println(
63                         "Error closing Reader associated with + \"" +
64                         file.getName() + "\".");
65                     e1.printStackTrace();
66                 }
67             }
68         } else {
69             System.err.println("Unable to open \"" + file.getName() +
70                 "\" for input");
71         }
72     }
73 
```

³Reader and Writer derived classes work with streams of `char`; there are `InputStream` and `OutputStream` classes which permit direct access to the stream of bytes. We note their existence here and return to working with streams of `char`.

74 `/**`

Listing 13.1: CountCharacters: count

CountCharacters is structured just like ExistsFile (see Listing 9.10) where the main method treats all command-line arguments as file names, constructs an object of the right type (CountCharacters in this case) with the file name and then calls the processing method, count. The field file is a File associated with the given file name.

count is twice as long as echo (Listing 9.11) and most of the code is part of the exception handling. Recall that when Java processes a `try...catch` statement the code in the `try` block is executed (lines 43-51) and if an exception is thrown, the first `catch` block following the `try` which matches the type of exception thrown will be executed.

Whether an exception was thrown or not, after finishing the `try...catch` block, the program will execute the `finally` block (if any) associated with the `try`. This makes the `finally` block a great place to close files. If the `new` in line 43 succeeded, reader is non-`null` and should always be closed. It is in line 62 which is only executed if reader was assigned a value in line 43.

Back to the `try`: just like a Scanner a FileReader can be constructed from a File object. Assuming no exceptions any open Reader provides a `read()` method which returns an `int`. The return type is `int` to accommodate Unicode returns. Each call to `read()` reads one character from the underlying stream, advancing the current position in the file, and returns the value. If the stream has been exhausted `read()` returns -1. This class defines a constant, EOF to make the loop on lines 46-50 easier to read.

If I run the program on its own input file, the following output is generated:

```
~/Chapter13% java core.CountCharacters core/CountCharacters.java
CountCharacters.java: 2427
```

Notice that the name of the class for Java to run is package name, a dot, and the name of the class. The name of the file which is a parameter to the program is the name of the folder, a slash, and the full name of the file.

When constructing a Scanner, if we pass it a File, under the hood, the Scanner constructs an `InputStreamReader`, the byte reading version of a `FileReader`. We can provide the Scanner with a `FileReader` to use as its input source.

```
38     FileReader reader = null;
39     try {
40         reader = new FileReader(file);
41         Scanner echoScanner = new Scanner(reader);
42         String line;
43         while (echoScanner.hasNextLine()) {
44             line = echoScanner.nextLine();
45             System.out.println(line);
46         }
47     } catch (FileNotFoundException e) {
48         System.out.println("PANIC: This should never happen!");
49         e.printStackTrace();
50     } finally {
51         if (reader != null) {
52             try {
53                 reader.close();
54             } catch (IOException e) {
55                 System.err.println(
56                     "Error closing Reader associated with + \"" +
57                     file.getName() + "\".");
58                 e.printStackTrace();
59             }

```

```

60     }
61     }
62     } else {
63         System.out.println("Unable to open \"" + file.getName() +
64             "\" for input");
65     }
66 }
67
68 /**

```

Listing 13.2: EchoFileWithReader: echo

EchoFileWithReader behaves just like Chapter 9’s EchoFile. The File is used to construct an FileReader which is then used to construct the Scanner. The FileReader is what is closed in the **finally** block; it is possible the reader was initialized and echoScanner was *not* so, to make sure the file is closed no matter what, we close the low-level FileReader if it is non-**null**.

FilterReader — Changing What Is Read

That is not very exciting. What if we wanted to *change* the value being echoed. That is, what if we wanted to convert curly braces (both open and close) into vertical bars, |? We could write a FilterReader-derived class.

The “Filter” in FilterReader is meant to make you think of a water filter and the stream of characters as the water passing through. Just as a stream of water may carry many different things along with it, some of which you would rather not see in your drinking glass, so might a character stream have characters we do not want passed to our program. For our example we want the stream modified so that no curly braces make it to our program and we want to introduce a different character at that same spot.

FilterReader is a Reader which takes a Reader as a parameter to its constructor. As written, every method defined in FilterReader just passes its parameters to the same method in the inside reader. These wrapper methods are provided so that child classes get all of the methods and can choose which ones they want to change.

```

14  /** state flag; have we already seen the end of the stream? */
15  private boolean endOfStream = false;
16
17  /**
24     this.endOfStream = false;
25  }
26
27  /**
35
36     ch = in.read();
37     if ((ch == '{') || (ch == '}')) {
38         ch = '|';
39     }
40     return ch;
41  }
42
43  /**
55     if (endOfStream) {
56         return -1; // end already reached
57     }
58
59     int charCount = 0;
60     for (int i = offset; i < (offset + length); i++) {

```

```

61     int temp = this.read();
62     if (temp == -1) {
63         endOfStream = true;
64         break;
65     }
66     text[i] = (char) temp;
67     charCount++;
68 }
69 return charCount;
70 }
71
72 /**
82     int charCountSkipped = this.read(chArray);
83     return charCountSkipped;
84 }
85 }

```

Listing 13.3: NoCurlyFilterReader

The `FilterReader` has a **protected** constructor. This is so that no one can construct one (this is similar to writing an **abstract class**), it is only possible to construct some subclass of `FilterReader` which declares a **public** constructor. `NoCurlyFilterReader` does that. It takes a `Reader` as its parameter and passes the object to `FilterReader`'s constructor. It also initializes a field which flags whether or not we have reached then end of the input stream. This will be explained below.

Lines 35-43 override the `read()` method. You can see, at line 38, that we call `read()` on the internal `Reader` provided as a **protected** field in `FilterReader`. If we didn't want to make any changes to the characters we could just pass `ch` back. Instead we check if the character is a curly brace. If it is, we set the value of `ch` to a vertical bar. Then line 42 returns `ch`.

There is one other `read` method we must override because it calls `in.read` directly (so it would not have any changed characters). That method is `read(char[] text, int offset, int length)`. The `[]` show that the first parameter is an *array*. `length` is the number of characters that should be read into the array and `offset` is the index where the first read character is to be stored. This is known as a *buffered read* where the array of characters is a buffer for holding some number of characters. This is the most general signature of a buffered read and while there are other buffered `read` methods, they are all defined in terms of this one. Another win for Do Not Repeat Yourself: because the most general version can do what all the others do (with the right parameters), it is the only one that needs to be written or, in our case, overridden.

Looking at the comments in the JavaDoc for `read`, we find

```

read
public abstract int read(char[] text,
                        int offset,
                        int length)
    throws IOException

```

Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.

Parameters:

- `text` - Destination buffer
- `offset` - Offsetset at which to start storing characters
- `length` - Maximum number of characters to read

Returns:

The number of characters read, or -1 if the end of the

```

    stream has been reached
Throws:
    IOException - If an I/O error occurs

```

The characters must be read into the array starting at `text[offset]` and continuing until `text[offset+length-1]`. The return value is the number of characters actually read into the array or, if the end of the input stream has been reached, `-1`.

This is where `endOfStream` comes in. When reading the last bunch of characters from the input, the buffered version of `read` will read in fewer than the required number. If the first call to `read()` (line 63) returns `-1`, then the buffered method returns 0 for the character count read. If, instead, the tenth call returns `-1`, then the buffered call returns 9. In any case, when the call to `read()` returns `-1`, indicating the end of the input stream has been reached, `endOfStream` is set **true**. Any future call to the buffered `read` will return `-1` (see lines 57-59).

So long as `read()` does not return `-1`, then the each entry in `text`, starting with `text[offset]`, is set to the next character read. To set an element of an array, the array name, the square brackets, and an index, appear on the left-hand side of an assignment operator as in line 68. The character count, the value returned, is incremented each time a character is added to the array.

This buffered version of `read` makes use of `read()` in line 63; this is used to read *every* character put in the buffer. This is important because it keeps us from having to repeat character replacement logic in the buffered `read` method. This exact buffered `read` code can work with multiple `read()` implementations (as we shall see below)⁴.

Line 82 begins the override of `skip(int)`. The `FilterReader` version of the method just calls **super**. `skip(n)`, returning the result. This method makes use of the buffered `read` methods which eventually calls the one defined at line 55. Because this filter replaces a single character with a different single character the number of characters read does not change so overriding `skip` is not, strictly speaking, necessary. Since `read()` could (and later, will) change the number of characters it reads and it returns, it is good practice to override `skip` as well as the general buffered `read`.

Decorators

A `FilterReader` or a class which **extends** `FilterReader` is an example of a recurring pattern in computer science, a pattern which comes up often enough to have its own name. These classes are *decorators*. A decorator is a class which wraps around an object (or a set of objects) of the same type.

One popular software design pattern book, *Head First Design Patterns*, by Freeman, *et. al.*[FFBD04], discusses decorators in terms of coffee drinks. The following discussion is loosely based on theirs (but much less complete).

A coffee drink is something you can purchase at the café. An **abstract** class, `CoffeeDrink`, represents *all* drinks. Part of the public interface of `CoffeeDrink` is `getPrice()` so that the system can determine what to charge customers.

When you order a drink you can request a flavor shot of chocolate, vanilla, or cinnamon. Any drink plus a flavor shot yields a *coffee drink*. The flavor shot is a *decorator*: it is both a coffee drink *and* a modifier of a coffee drink. It is possible that the coffee drink being decorated is another flavor shot.

Look at the inheritance diagram, Figure 13.1. The three coffee drinks, `Coffee`, `Espresso`, and `FlavorShot` are shown, each extending the **abstract** class `CoffeeDrink`. Each box represents a class and each arrow represents the **extends** relationship, pointing at the child class.

When ordering an Espresso, the system constructs a `Espresso` object. This situation is shown in Figure 13.2(a). What happens when you get a `Coffee` with a chocolate `FlavorShot`? Just like the barista makes the coffee and then augments it with the flavoring the system constructs a `Coffee` object and passes that object into the constructor for a `FlavorShot`. The resulting composite object is seen in Figure 13.2(b). Finally, when a customer orders an espresso with vanilla and cinnamon, the result is seen in Figure 13.2(c).

⁴In fact, the implementations of `read(char[], int, int)` and `skip(int)` are slight variations on versions written for a completely different `FilterReader` project in *Java I/O, 2E* by Harold [Har06].

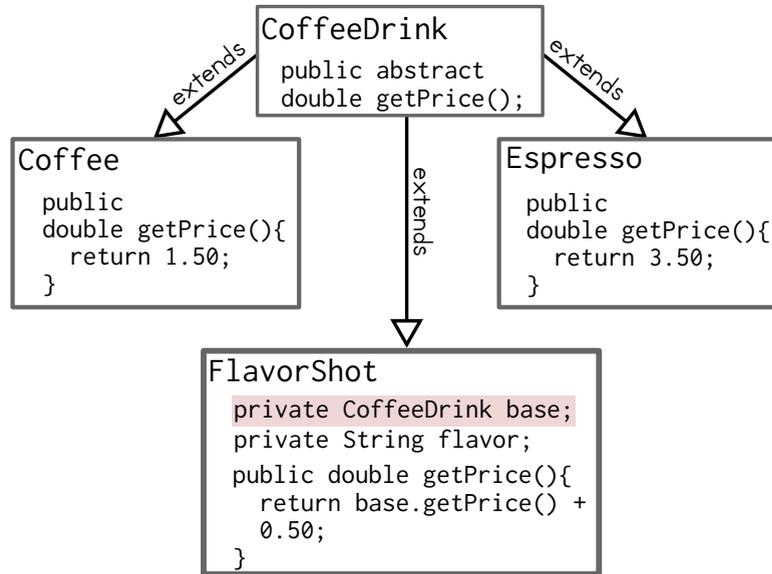


Figure 13.1: CoffeeDrink Class Hierarchy

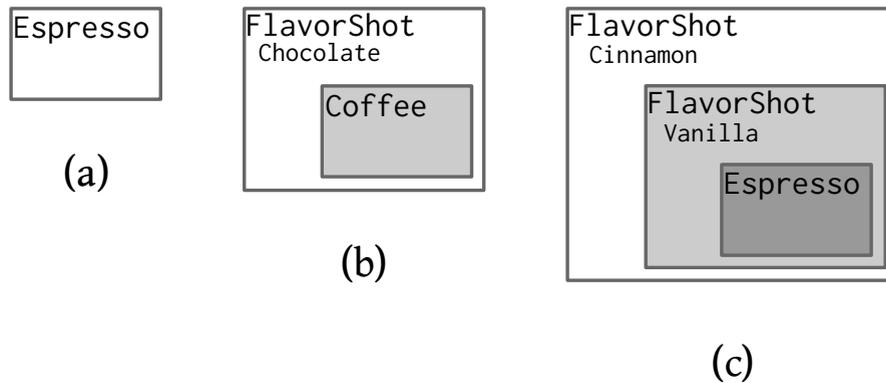


Figure 13.2: Decorated CoffeeDrink Objects

When checking out, the system asks the `CoffeeDrink` object to `getPrice()` so it knows how much to charge. For object (a) in the figure `Espresso.getPrice()` is called and returns 3.50. For object (b) `FlavorShot.getPrice()` is called. Notice that the first thing it does is call `getPrice()` for *whatever drink* the flavor shot was based on. This calls `Coffee.getPrice()` which returns 1.50. When `Coffee.getPrice()` returns, `FlavorShot.getPrice()` adds 0.50 to the value and returns it for a price of 2.00.

The double-flavored espresso in (c) is handled similarly with the sequence of calls going from `FlavorShot.getPrice()` (cinnamon) to `FlavorShot.getPrice()` (vanilla) to `Espresso.getPrice()`. These return, in reverse order, 3.50, 4.00 (vanilla), and 4.50 (cinnamon). The flavors in parentheses are just to differentiate the two flavor shots (to show the last-in-first-out nature of the calls).

Back to our `FilterReader`. Notice that `read()` does call the inner `Reader`'s `read()` method. This is a common aspect of a decorator: it wraps around some object, providing the same public interface, and the routines implementing the interface call into the wrapped object. The decorator does more than just forward calls,

though. It also decorates the results or changes the values coming back. By implementing the same public interface (either by extending the same object or literally implementing the same Java `interface`), the decorated object can be used in all the same places the undecorated object could be used.

```

41     Reader reader = null;
42     try {
43         reader = new NoCurlyFilterReader(new FileReader(file));
44         Scanner echoScanner = new Scanner(reader);
45         String line;
46         while (echoScanner.hasNextLine()) {
47             line = echoScanner.nextLine();
48             System.out.println(line);
49         }
50     } catch (FileNotFoundException e) {
51         System.out.println("PANIC: This should never happen!");
52         e.printStackTrace();
53     } finally {
54         if (reader != null) {
55             try {
56                 reader.close();
57             } catch (IOException e) {
58                 System.err.println(
59                     "Error closing Reader associated with + \"" +
60                     file.getName() + "\".");
61                 e.printStackTrace();
62             }
63         }
64     }
65     } else {
66         System.out.println("Unable to open \"" + file.getName() +
67             "\" for input");
68     }
69 }
70
71 /**

```

Listing 13.4: EchoFileWithFilterReader: echo

The `echo` method of `EchoFileWithFilterReader` is, except for line numbers, identical to the `echo` method of `EchoFileWithReader` except for the line assigning a value to `reader` (line 45 here, line 42 there). Here the `reader` is a `NoCurlyFilterReader` wrapped around a `FileReader`. The `Scanner`, written to the interface, does not care that the object type is different. It reads the file which is decorated by the `FilterReader` so that all curly braces are replaced with vertical bars.

It is possible to decorate something by removing something as well as by adding something. We will now explore how to write a `FilterReader` which removes end-of-line comments from a text file as it reads it. We will also discuss why that would be a good thing for writing a self-describing file reader.

Preprocessing Comments

The first step to compiling a Java program is to break it down into *tokens*. A token is a string that has meaning to Java: keywords, operators (+, =, etc.), punctuation ({, }, ;, etc.), integer and double literals, and string literals. You should convince yourself that processing the stream of input characters to tokenize the source code is not a trivial task (consider how you would handle quoted strings with escape characters or even identifiers). Imagine, while tokenizing, you also had to juggle comments: when you're inside a comment, the sequence "int" means nothing; when outside comments, "int" is a keyword.

The Java tokenizer would be much simpler to write if only programmers would leave out comments⁵. If comments are necessary, perhaps we could split the tokenizing task in two: first process the stream of characters into another stream containing only those characters not in comments. That is, *filter out* the comment text. The tokenizer is then much easier to write and the two processes together, comment stripper and tokenizer, are probably simpler to write and understand than a comment-avoiding tokenizer⁶.

It is similarly easier to preprocess a data file to remove comments and then process the comment-free records rather than read information with a `Scanner` and have to worry about inside/outside comment. A `FilterReader` can process an incoming stream of characters into a different sequence of characters, filtering out all of the comments in the self-describing data file.

To further simplify the job of stripping comments we will only support end-of-line comments. So, how can you *filter out* Java-style end-of-line comments?

Given a `Reader` reading the following character sequence:

```
Line one // beginning
// this whole line is blank
three/visible right here
//
Line five (blank above^)
```

we want to process it into the following character sequence:

```
Line one

three/visible right here

Line five (blank above^)
```

`SansCommentFilterReader` is a `FilterReader`. It scans its input for the `//` sequence and skips over the characters from the first slash to the end of the current line. The characters are skipped by reading them from the input `FileReader` but *not* returning any of them as the result of calling `read`. End-of-line comments will be “replaced” with the end-of-line marker.

Figure 13.3 shows our file on disk being read by a `FileReader`. The sequence of characters read is shown on the arrow going into the `FilterReader`. The sequence returned, one after the other, by `FileReader.read()` is shown on the arrow going into `SansCommentFilterReader`. The sequence in and the sequence out of `FileReader` are identical.

The arrow going from the `SansCommentFilterReader` to the `Scanner` shows the characters returned by `SansCommentFilterReader.read()`. The `Scanner` is performing `next`, `nextLine`, `skip`, and other methods reading the information. The sequence of characters it gets back from the sans comment filter is no longer (and in this case, *shorter*) than the number of characters the sans comment filter gets back from the file reader.

`SansCommentFilterReader` is identical to `NoCurlyFilterReader` except for `read()` so we will examine that method.

```
27  /**
28   * Read one character while replacing '{' and '}' with '|'
50   * @return number of char read or -1 if past end of stream
51   */
52  @Override
53  public int read(char[] text, int offset, int length)
54      throws IOException {
55      if (endOfStream) {
56          return -1; // end already reached
```

⁵Most students would be more than happy to oblige.

⁶This is how most programming languages are compiled. An early phase goes through and removes comments from the sequence of characters and only later is the sequence of characters changed into a sequence of tokens.

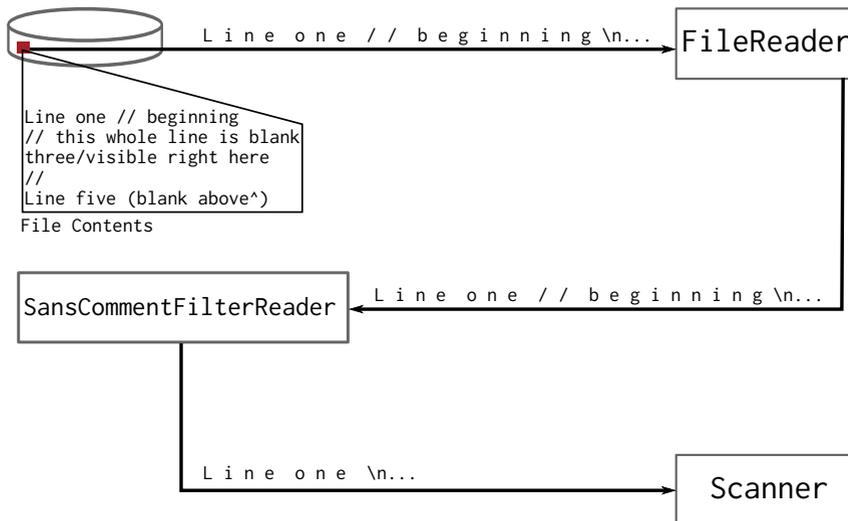


Figure 13.3: A sequence of FilterReaders.

```

57     }
58
59     int charCount = 0;
60     for (int i = offset; i < (offset + length); i++) {
61         int temp = this.read();
62         if (temp == -1) {
63             endOfStream = true;
64             break;
65         }
66         text[i] = (char) temp;
67         charCount++;
68     }
69     return charCount;
70 }
71
72 /**
73  * Skip over the given number of characters.
74  *
75  * @param n number of characters to skip over
76  *
77  * @return number of char skipped or -1 if past end of stream
  
```

Listing 13.5: SansCommentFilterReader

There is one new field, `readAheadCh`. When reading a `'/'`, it is necessary to read the next character to see if the character marks the beginning of a comment. This is what happens in the first line in the data file above. The next character is `'/'` so we skip a comment. It happens again in the second line. The `read()` method sees a `'/'` followed by a `'v'`. We have a problem. This time through the slash character should be returned. But we have already read the next character, the `'v'`; how can we remember that we read a good character and return

it the *next* time `read()` is called. `readAheadCh` stores the value of any character read accidentally. It is set to `-1` (in the constructor and in `read()`) when it does not contain a useful character.

So, lines 55-59 handle returning the saved value (and resetting `readAheadCh` to `-1`) when necessary. When the method **returns** in line 58 execution of the method stops (so line 61 is never reached in that case).

If there is no read ahead character to return, we decorate the value returned by the Reader in line 61. If `ch` is anything other than a slash, then it is just returned. If it is a slash, we must read ahead. If the read ahead is *not* a slash, the slash is the right character to return and we save the read ahead.

If the read ahead was a second slash then we skip over the rest of the current line. A line is ended by a `'n'` (newline), a `'r'` (a carriage return) or a combination of the two characters. We don't have to worry about how the current line is terminated: just use a sentinel-controlled loop to read until one of the end-of-line characters is seen and return that character. This puts the end-of-line marker exactly where the first slash was (to anyone watching just the decorated output).

In `SansCommentFilterReader` it is necessary to override `skip` because the number of characters read from `in` does not relate to the number of characters returned from `read()`. `skip` should skip over characters which would have been returned to the user of the Reader.

Attribute-Value Pairs

An *attribute-value pair* is an abstract way of encoding description information. An *attribute* is some aspect of a game object expressed as a `String` naming the attribute: `Age`, `Class-Number`, `CollegeIDNumber`. Each attribute has, associated with it, a *value*. The value is also expressed as a `String` which we can, if necessary, interpret as a number or some other object: `29`, `CIS 201`, `P005-555-1212`. This is a flexible way of storing information which we can adapt to storing fields of objects. Each field is stored as an attribute with the name of the field and a value of the value intended for the field. It is also possible to use attribute-value pairs to store command aliases for the text adventure game.

Command Aliases

Consider processing the various commands the player types into a text adventure game. With compass directions as movement commands, a player would type "north" to go north. The next time they want to move they would, again, type "north". And so on. Think about the code you would use to process this. Assume there are `go<Direction>()` methods which handle movement.

```
if (command.equalsIgnoreCase("north")) {
    goNorth();
} else if (command.equalsIgnoreCase("south")) {
    goSouth();
} else if (command.equalsIgnoreCase("east")) {
    goEast();
} else if (command.equalsIgnoreCase("west")) {
    goWest();
} else {
    unknownCommand();
}
```

This snippet assumes the movement directions are the only possible commands; in the real game there will be commands like "get" and "put", perhaps "attack" or "talk". Consider the user's experience typing the movement commands. "north" is only five characters but typing it over and over is a waste of the player's time. It would be nice if "n" was enough.

So, how would you change the first Boolean expression in the snippet to handle either "n" or "north"?

```
if (command.equalsIgnoreCase("north") ||
    command.equalsIgnoreCase("n")) {...
```

That would then need to be repeated for each of the directions. But what if our game has a convention that the user is always facing north and we want “forward” and “F” to be *aliases* for “north” (or “n”). An alias is a different name for the same thing. We want to support an arbitrary number of aliases for each command.

Putting the aliases in the code violates the concept of separating program and data. Some number of basic commands will have to be coded in the game but aliases should be more flexible: it should be possible to add an alias for an existing command without having to recompile the code.

A file full of alias-command pairs (where the alias is the attribute and the command is the value) could be read into a dictionary. Then, instead of rewriting the `if` statement when we wanted new aliases, we can just update the alias file used to initialize the dictionary.

Assume the variable `dictionary` is a `Dictionary` which has been filled from the alias file. `Dictionary` has a `get` method which, given an attribute, will return the corresponding value. It will return `null` if there is no match. Then we could rewrite the code snippet above to

```
String normalizedCommand = dictionary.get(command);
if (normalizedCommand.equalsIgnoreCase("north")) {
    goNorth();
} else if (normalizedCommand.equalsIgnoreCase("south")) {
    goSouth();
} else if (normalizedCommand.equalsIgnoreCase("east")) {
    goEast();
} else if (normalizedCommand.equalsIgnoreCase("west")) {
    goWest();
} else {
    unknownCommand();
}
```

The data file would look like this:

```
n=north
north= north
// more directions follow
s =south
south = south

e =<east>
east    =<
    east> // and so on
// western stuff here
west = <west >
w = < west >
```

The odd spacing and confused ordering are on purpose. We need to deal with free-form files so being able to read a file in this form will make reading objects simpler.

Dictionary

The `Dictionary` class has the following public interface:

```
public class Dictionary {
    public Dictionary()...
    public Dictionary(Scanner dictionaryFile)...
    public String get(String attribute)...
    public boolean hasKey(String attribute)...
    public String put(String attribute, String value)...
```

Another name for an attribute in a dictionary is the *key*, the special value used to look up an entry. This interface is modeled on the Map interface provided in `java.util`; Map is part of the Java *Collections* package like List and ArrayList. That is why the method to test if a given attribute is in the Dictionary is called `hasKey`.

The first constructor builds an empty Dictionary. The second reads a file with lines of the form `<alias>=<command>`. The file can have comments if the Scanner is constructed with a `SansCommentFilterReader` and it can contain blank lines which will just be ignored. Extra blank space around the equal sign is also ignored.

The `get` method looks up the entry in the Dictionary. If there is an entry, the value is returned. If no such entry exists, `get` returns `null`. `hasKey` returns `true` if there is an entry with the given attribute and `false` otherwise. `put` associates the given value with the given attribute; `put` returns the old value of the attribute or `null` if it had no previous value.

The entries in the Dictionary are `AttributeValuePair` objects stored in the `allEntries` ArrayList. The public interface for `AttributeValuePair` is

```
public class AttributeValuePair
implements Comparable<AttributeValuePair> {
public static String beforeEquals(String line)...
public static String afterEquals(String line)...
public AttributeValuePair(String line)...
public AttributeValuePair(String attribute, String value)...
public boolean compareTo(AttributeValuePair rhs)...
public String getAttribute()...
public String getValue()...
public String setValue()...
public String toString()...
```

The two `static` methods are `public` because other methods might be able to use them. They are utility functions which split a line on the left-most equal sign. They handle lines without any equal sign by assuming it all comes before an equal sign so `before` returns `line` and `after` returns `""`.

The second constructor is the primary constructor: given an attribute and a value it fill the fields in the object. The first constructor uses `beforeEquals` and `afterEquals` to extract the parameters for calling the second constructor. The getter/setter method are standard except that there is no `setAttribute` method. Once a pair is made, the attribute is *immutable*: it cannot be changed. It makes sense to keep it fixed because comparison is by attribute only (`compareTo` returns the result of comparing the attribute fields of the two `AttributeValuePair` object). If the attribute could change, then the sorted order of a list of `AttributeValuePair` objects could be modified by assignment to one of the elements of the list.

```
20 public static void main(String[] args) {
21     if (args.length > 0) {
22         Scanner aliases = null;
23         try {
24             aliases = new Scanner(new SansCommentFilterReader(
25                 new FileReader(new File(args[0]))));
26             Dictionary dictionary = new Dictionary(aliases);
27             System.out.println("=====");
28             System.out.println("dictionary = \n" + dictionary);
29             System.out.println("=====");
30         } catch (FileNotFoundException e) {
31             System.err.println("Unable to open " + args[0] + " for input.");
32         } finally {
33             if (aliases != null) {
34                 aliases.close();
35             }
36         }
37     }
38 }
```

```

37     } else {
38         System.err.println("usage: java TestDictionary dictionaryFileName");
39         System.err.println("        where dictionaryFileName names a file");
40     }
41 }

```

Listing 13.6: TestDictionary main

core.TestDictionary constructs a Dictionary with a file name provided on the command-line and then prints the contents of the dictionary to standard output. When running it with the sample aliases given earlier, the output is:

```

~/Chapter13% java core.TestDictionary aliases.txt
=====
dictionary =
e = east
east = east
n = north
north = north
s = south
south = south
w = west
west = west
=====

```

This is what we would expect because the Dictionary was described as being kept in sorted order. We will look at how Dictionary.get takes advantage of this ordering in Section 13.5.

Reading Game Objects One Field at a Time

How could we use the comment stripping and attribute-value pairs reading to read a self-describing record of some class type? Comment stripping makes it possible to support commented files. Attribute-value pairs can be used to implement order independence: each field will be stored with its name, an equal sign, and then the value. An object factory needs two pieces of information we have not yet accounted for: the type of the object to create and when the data for that object ends. The two will be marked similarly: the name of the class will appear on a line by itself, then any fields as attribute-value pairs, and then the name of the class with a slash (/) in front of it. Thus a Location could be stored like this:

```

Location // First room in the registrar's office
uuid = Registrar:Reception
name = Registrar's Reception Office
lights = off // so user can't see the problem
description = There is a white desk and a bulletin-board.
dark-description = You can't see a thing. Turn on lights.
// uuid of Location in direction; not here = no Location
north = Registrar:MeetingRoom
south = Admin:CopyRoom
east = Admin:Atrium
/Location

```

This self-describing file tells us the names of the fields in the Location class (some of which are common to all GameObjects). Looking at the two different kinds of descriptions, this file format looks constrained: the description is the text of the story in a text adventure game and limiting it to the end of a single line is too great of a constraint. While Java (and many text editors) have no problem with lines of arbitrary length, it often hard to edit long lines. We need some way to designate a multi-line value for an attribute.

Parsing Multi-line Values

A container, like a Java block, makes sense. If the first character in a value is the container opening character, then the value continues until the corresponding close character is seen; if the first character is anything else, then the value continues to the end of the current line. The only question now is what characters to use to set off our value *block*. For ease of processing we will not support nested containers (no blocks within blocks), nor will we support having close characters inside the value (there will be no escaped characters).

Using the new notation, a more complete description (and dark-description) can be written:

```
Location // First room in the registrar's office
uuid = Registrar:Reception
name = Registrar's Reception Office
lights = off // so user can't see the problem
description = <There is a tall white desk facing the door.
Behind the desk, reaching to the ceiling, is a pink bulletin
board. It is covered with ponies, unicorns, and rainbows.
The sweetness is almost enough to induce a diabetic coma.>
dark-description = <The darkness fairly pulses with... something.
It is far too dark to see anything and your other senses seem
almost afraid to detect anything. The light switch is probably
on the wall to the left.>
// uuid of Location in direction; not here = no Location
north = Registrar:MeetingRoom
south = Admin:CopyRoom
east = Admin:Atrium
/Location
```

We need to break an attribute-value pair into three pieces: the attribute, the equal sign, and the value. The attribute is a run of non-whitespace characters terminated by a space or by an equal sign. The equal sign is just that, the string "=" but it can appear following any amount of white space. Finally, the value is either any run of non closing angle bracket characters enclosed between angle brackets, "<" and ">" or the remainder of the current line after the "=".

The read* Helper Methods

The three helper functions for reading these parts of an attribute and value are more general than just reading an attribute-value pair. They will go in a utility class which will also hold some output helper methods. Thus the class is `util.ReadAndWrite`. The methods will be **static** methods returning `String` read from a provided `Scanner`.

```
20  *
21  * @param in the {@link Scanner} from which to read
22  *
23  * @return next attribute token: skip whitespace,
24  */
25  public static String readAttribute(Scanner in) {
26      String retval = "";
27      in.skip("\\s*");// unlimited whitespace
28      Pattern oldDelimiter = in.delimiter();
```

Listing 13.7: `ReadAndWrite` `readAttribute`

`readAttribute` uses a *regular expression pattern* to skip over whitespace. A regular expression is a compact expression for a pattern of characters which can be interpreted by a regular expression matcher. Regular expressions, shortened as *regexes*, use a simple, formal language to express character sequences. The `Scanner` class has several methods with use regexes: `findInLine`, `findInHorizon`, and `skip`. Earlier we used the pattern

string ". *"; the . is a regex matching any single character and the * modifies any preceding regex to match 0-or-more copies of itself.

What does the pattern string on line 22, "\s*" match? \s is a pattern matching any whitespace character (a space, end-of-line marker, tab, or Unicode equivalents). The extra \ is necessary so that the string passed in to skip is \s (the backslash is an escape character in a String so to get a backslash character in a String the escape must be escaped: "\" is the single character String containing a backslash). As stated above, the * modifies the regex before it to match zero or more copies of itself. Thus the pattern is any number of whitespace characters.

When Scanner.next is called, it skips over any number of copies of a pattern called the *delimiter*. The default delimiter is \s, any whitespace character. After finding a character not in the delimiter, next collects up all of the characters until it finds another delimiter match. This more detailed view explains how next, with the default delimiter, skips over all whitespace and returns one word at a time. A word is any contiguous sequence of non-whitespace characters.

It is possible to change the delimiter used by a Scanner. This can be useful for reading specially formatted input like our attribute. The description above says an attribute ends with whitespace or an equal sign. The pattern string in line 24, "[\s=]" is a pattern consisting of whitespace or =. The square brackets are a regular expression matching any character in the list of characters in the brackets. So \s is the list of whitespace and = is that character. Thus this delimiter matches whitespace or equals. Thus the call to next in line 25 stops where it should. Notice that we already skipped over any leading delimiter so we know we are looking at a non-whitespace character. Lines 23 and 26 are required so that the call to readAttribute does not change how the Scanner normally reads the input file.

```

51 * @param in the {@link Scanner} from which to read
52 * @param match the string to match
53 *

```

Listing 13.8: ReadAndWrite readMatch

readMatch takes a Scanner and a String to match. The findInLine method takes a pattern string and, ignoring the delimiter completely, searches for a match from the current read point. If a match is found before the end of the current line in the stream, then the matching string is returned and the current read point is moved past the match. No match means it returns `null` and the current read point is unmoved.

For finding the =, we just pass in the pattern string "= ". This will match the equal sign and the Scanner will skip over anything before it finds a match. Thus any whitespace to the left of = is skipped by this method.

```

72 *         read position to end of the current line if next character
73 *         is not '{'; the value between (but not including) ' {' and
74 *         ' }' otherwise. Note: it goes to the very next ' }' so it
75 *         does not support escapes or nested groups.
76 */
77 public static String readValue(Scanner in) {
78     String retval = "";
79     String openMark = readMatch(in, "<");
80     if (openMark == null) {
81         retval = in.nextLine();
82     } else {

```

Listing 13.9: ReadAndWrite readValue

readValue uses matchString to figure out if it is looking at an angle brace or something else. This makes use of how Scanner.findInLine works when the pattern matches and when it fails to match. If "<" matches, then the current read point is just past the < character. That means it is at the beginning of the contained text, the value that should be read. If the pattern fails to match, the value returned is `null` (for the if statement) and the current read pointer remains *unchanged*.

If `openMark` is `null` (no match), then the return value is just the rest of the line as read with `nextLine`. If `openMark` is non-`null` (a match), then the return value is set using `findWithinHorizon`. `findWithinHorizon` is like `findInLine` except it searches for some number of characters rather than until an end-of-line marker.

The use of 0 for the horizon means search for the rest of the file. The pattern string, "`[^>] *`" means 0-or-more characters which are *not* `>`. The `[^` as the opening of the list of characters means "all characters except the following" so the pattern `[^>]` is any one character that is not a closing angle bracket. The `*` means any number (including 0).

The skip in line 79 uses the pattern string to skip over the `>` and any end-of-line markers following it.

Each Tending to its Own

A factory is not too hard to write now. We will present the `Location` factory because it is the first we will use when we write our first phase of the game, a game with just a map you can move on. The overview is that we will read the name of the class, then handle each attribute-value pair until we see the end of class name.

How is "handle each attribute-value pair" done? The problem is, who knows how to handle the name (attribute) of all the **private** fields in `Location`, `GameObject`, `Item`, and `Critter`? No one object has access to all of those fields. But that is actually a good thing. We will pass the attribute-value pair into the lowest element in the hierarchy and that object will *first* pass the pair up to its **super** class. If the **super** recognizes the attribute, it handles it; if the **super** fails to recognize it, then the child tries to recognize it, returning **true** if it recognizes it and uses the associated value to set a field. The returning **true** is important because that is how a child of the child would know the field was handled.

```

59 public static Location readObjectFromFile(Scanner gameObjectScanner) {
60     String classID = gameObjectScanner.next();
61     if (!classID.equalsIgnoreCase("Location")) {
62         return null; // not the expected type of object; punt
63     }
64
65     String endClassID = "/" + classID;
66
67     Location lo = new Location();
68     GameObject.processAttributes(gameObjectScanner, endClassID, lo);
69     return lo;
70 }

```

Listing 13.10: `Location` `readObjectFromFile`

The method assumes it is called with the current read point somewhere before the opening `Location` line in a location record file. Line 60 reads the next word from the file; since the file is opened with a `SansCommentFileReader`, the next word should be the name of the class. This factory, rather than handling multiple different class names, validates that the class name found in the file matches "Location". If it doesn't match then there is no point in reading the record so we return `null`. If the `Location` constructor is called, then the `processAttributes` method is called.

```

95 protected static GameObject processAttributes(
96     Scanner gameObjectScanner, String endClassID, GameObject go) {
97     String attribute = ReadAndWrite.readAttribute(gameObjectScanner);
98     while (!attribute.equalsIgnoreCase(endClassID)) {
99         /* ignore */ ReadAndWrite.readMatch(gameObjectScanner, "=");
100        String value = ReadAndWrite.readValue(gameObjectScanner);
101        go.handleAttributeValuePair(attribute, value);
102
103        attribute = ReadAndWrite.readAttribute(gameObjectScanner);
104    }
105    return go;

```

```
106 }
```

Listing 13.11: GameObject processAttributes

In processAttributes the three reading helpers are used. The first reads the attribute and if the attribute is the end-of-record string, then the record is done so the method returns. Otherwise the attribute and value are passed to the object's handleAttributeValuePair; this method is overridden in Location (and Critter and Item) so the call is *dynamically* made to Location.handleAttributeValuePair.

```
257 protected boolean handleAttributeValuePair(String attribute,
258     String value) {
259     if (super.handleAttributeValuePair(attribute, value)) {
260         return true;
261     }
262
263     if (attribute.equalsIgnoreCase("lights")) {
264         setLights(value.equalsIgnoreCase("on"));
265         return true;
266     } else if (attribute.equalsIgnoreCase("darkdescription")) {
267         setDarkDescription(value);
268         return true;
269     } else if (attribute.equalsIgnoreCase("north")) {
270         setNorth(value);
271         return true;
272     } else if (attribute.equalsIgnoreCase("south")) {
273         setSouth(value);
274         return true;
275     } else if (attribute.equalsIgnoreCase("east")) {
276         setEast(value);
277         return true;
278     } else if (attribute.equalsIgnoreCase("west")) {
279         setWest(value);
280         return true;
281     }
282
283     return false;
284 }
```

Listing 13.12: Location handleAttributeValuePair

The first thing that happens, line 259, is that the GameObject version of this method is called. Thus GameObject gets first shot at handling the given attribute name. If it does (we will see the code below), it returns **true** and the Location version is done: it just returns **true** to signal that the field was consumed somewhere at or above Location.

If the method reaches line 263, this is a large **if/else if** structure. It checks the attribute string against the names of fields it knows. If there is a match, the value is interpreted (as the appropriate type) and the method returns **true**.

The code in GameObject.handleAttributeValuePair is very similar to that in Location, just without the call to the **super** version of the method (there isn't one).

```
316 protected boolean handleAttributeValuePair(String attribute,
317     String value) {
318     if (attribute.equalsIgnoreCase("uuid")) {
319         setUUID(value);
320         return true;
```

```

321     } else if (attribute.equalsIgnoreCase("name")) {
322         setName(value);
323         return true;
324     } else if (attribute.equalsIgnoreCase("description")) {
325         setDescription(value);
326         return true;
327     }
328     return false;
329 }

```

Listing 13.13: GameObject handleAttributeValuePair

The three fields that it “knows” are `uuid`, `name`, and `description`. Thus if the attribute contains any of the three, the value is assigned to the right field. The return value of `true` tells `Location.handleAttributeValuePair` that the attribute has been handled.

13.4 Incremental Development

This program has more packages than `Tetris` and a few more lines of code. When writing a program of more than trivial length, how should you approach development? What classes and methods should you write first?

This section will take a quick diversion to talk about a software engineering approach that has much currency as this book is going to press, the *agile* approach. The agile approach focuses on breaking the final project into units value to the user of the system. The programmer then focuses on implementing units of value in an order determined by the intended customer (best return on investment first).

The Agile Approach

The agile approach is spelled out in the *Agile Manifesto*[Gro07] which cites a dozen principles. Many of them have more to do with team projects developed directly for a customer, things beyond the scope of this book. We will focus on a four of them in this section; the following are paraphrased from the *Manifesto* itself.

- Simplicity is essential: maximize what is *not* done.
- Deliver working software frequently.
- Working software is the primary measure of success.
- Continuous attention to technical excellence enhances agility.

Hopefully the first item seems familiar. Simplification is one of the goals of the book, a theme underlying the idea of using the right level of abstraction.

The idea of delivering working code frequently is what motivated this section. We just spent a long, long section discussing how to read structured text files. While reading text files is necessary to make a text adventure game, a program which just reads structured text files is not very interesting.

The measure of success is delivering working code. The key is that the definition of *working* for our purposes is delivering a game. Perhaps a simple, not quite complete game, but a game all the same.

The agile approach, because of the focus on working code, is considered at odds with the *waterfall model*, a model where very complete documentation of the interfaces and implementations is done before coding. Some students reading that working code is the most important thing begin to wonder why they waste time on writing comments. Comments are not code!

But comments are necessary in excellent code. Agility requires delivering code in multiple phases. That means the code will be read by the agile programmer over the course of development as it is enhanced in each phase. Thus comments are part of technical excellence and make the code maintainable (sustainable coding practices and pacing are another of the dozen principles).

This is a very cursory introduction to the idea of agile development. There are many books and web sites devoted to different methodologies that support the agile approach. The next section will look at how to break our text adventure game into phases, phases which are, more or less, games.

Delivering Units of Work

What is the least functionality we could deliver and consider what we have a partial text adventure game? A dictionary test program? No, while the alias dictionary is a useful tool, it is not, by itself, a game.

What about a program that reads a file full of locations? That would test out the `GameObject` reading code. What will the program *do*?. How about a program which read the map file and permits the player to move around the map? That would let us test the reading code *and* deliver a program that is a step along the way to a text adventure game.

Loading the Whole Map

Given that we have looked at the code for reading a single record from a `Location` (or any other `GameObject`) file, how will the game keep track of the whole map. We have one answer when it comes to storing a collection of objects: an `ArrayList`. The code to read the whole list can be broken into two methods, one to open the file and one to read it. The opening code is broken out because we will reuse it when reading the `Critter` and `Item` files.

```

158 private Scanner openFileForInput(String fname) {
159     Scanner opened = null;
160     try {
161         opened = new Scanner(new SansCommentFilterReader(
162             new FileReader(new File(fname))));
163     } catch (FileNotFoundException e) {
164         // do nothing (null will be returned which is what we want)
165     }
166     return opened;
167 }
202 private ArrayList<Location> readMapFile(String fname) {
203     ArrayList<Location> returnMap = null;
204     Scanner mapFile = openFileForInput(fname);
205     if (mapFile != null) {
206         returnMap = new ArrayList<Location>();
207         while (mapFile.hasNext()) {
208             Location location = Location.readObjectFromFile(mapFile);
209             if (location != null) {
210                 returnMap.add(location);
211             }
212         }
213     }
214     return returnMap;
215 }

```

Listing 13.14: `GameWithMapOnly` Map Reading Routines

The `openFileForInput` takes the name of the file and returns a `Scanner` wrapped around a `SansCommentFilterReader` which is, in turn, wrapped around a `FileReader` reading the given file. Thus when `readMapFile` uses the `Scanner`, comments are automatically skipped (never to be seen). Because opening the file is more complex than just calling `new Scanner`, it makes sense to break it out into its own method. Further, notice that the method returns `null` if there is a problem (*i.e.*, an `Exception` is thrown). Thus the `read` method can tell there was a problem and return `null` as the value for the.

The `read` method uses an eof-controlled while loop: it checks if there is a next token (non-blank space) and if there is it tries to read a `Location`. If there was no problem reading the `Location`, then the new `Location` is added to the map. Finally the map is returned. In the constructor for a `GameWithMapOnly`, the `map` field is set to the result of the `read` method.

```

56 public GameWithMapOnly() {
57     aliases = new Dictionary(openFileForInput("aliases.txt"));
58     map = readMapFile("dwarf.loc");
59     if (map != null) {
60         player = map.get(0).getUUID();
61     }
62 }

```

Listing 13.15: GameWithMapOnly Constructor

The `aliases` field is also initialized in the constructor. What is line 60 all about? We need some way to keep track of where the player is in the game. Since we have no `Critter` objects we cannot use one to keep track of the player. So, we can keep track of the player by having a `String` field, `player` that holds the UUID of the location where the player is. Line 60 makes the arbitrary decision that the player begins in the first location in the location file (the one put at index 0 in `map`). This is arbitrary but since we are just moving around in the map, it is as good a place to start as any other.

Moving

So, what does the game loop look like? It needs to show the current state of the game: this is just printing out the `Location` where the player is. Then it needs to get input from the user: we will define a `Keyboard` class which wraps a `Scanner` around `System.in` and provides `next`, `nextLine`, `nextWithPrompt`, and `nextLineWithPrompt`. It will also have versions of the old `answeredYes` method. All the methods are **static**. Finally, the state of the game must be updated: we will use the multi-way **if** described to motivate the `Dictionary` adding just an exit command to permit the player to exit the game.

```

67 public void play() {
68     gameOver = false;
69     while (!gameOver) {
70         // show state
71         System.out.println(findLocation(player));
72         // get user input
73         String command = Keyboard.next();
74         // update state
75         processCommand(command);
76     }
77 }

```

Listing 13.16: GameWithMapOnly play

`play` is the main game loop. The location referred to by `player` is looked up (we will examine how it is looked up in the next section) and the whole thing is displayed. The `toString` method dumps all of the field values:

```

~/Chapter13% java core.GameWithMapOnly
Game Begins
gamestuff.Location [
uuid = Dwarf:Workshop
name = Dwarf's Workshop
description = The ceiling is low, the soot-stained rock forcing you
to bend almost double. The heat from the banked forge in the
north-east corner of the room hits you like a hammer. It is hard to
imagine any creature surviving when it is at full blast.
darkDescription = There is a diffuse, orange glow from the north-east, a
smell of burning rock and metal. The orange is filled with heat.

```

```
lights = true
north = Dwarf: Bedroom
south = Dwarf: Hall: North
east = NO_SUCH_LOCATION
west = NO_SUCH_LOCATION
]
```

The values are those read in from `dwarf.loc` in the first position. The cursor is waiting on the line below the listing for a command to be typed.

```
177 private void processCommand(String command) {
178     String normalizedCommand = aliases.get(command);
179     if (normalizedCommand.equalsIgnoreCase("north")) {
180         goNorth();
181     } else if (normalizedCommand.equalsIgnoreCase("south")) {
182         goSouth();
183     } else if (normalizedCommand.equalsIgnoreCase("east")) {
184         goEast();
185     } else if (normalizedCommand.equalsIgnoreCase("west")) {
186         goWest();
187     } else if (normalizedCommand.equalsIgnoreCase("exit")) {
188         gameOver = true;
189     } else {
190         unknownCommand(command, normalizedCommand);
191     }
192 }
```

Listing 13.17: `GameWithMapOnly` `processCommand`

When the command is typed, it is passed in to `processCommand` which looks it up as an alias. So, if the player enters `south`, the alias for that command was also `south`. In `processCommand`, `south` matches in line 181 so `goSouth` is called.

```
126 private boolean goSouth() {
127     Location here = findLocation(player);
128     boolean canMove = (here.getSouth() != Location.NO_SUCH_LOCATION);
129     if (canMove) {
130         player = here.getSouth();
131     }
132     return canMove;
133 }
```

Listing 13.18: `GameWithMapOnly` `goSouth`

How can we move south? First we check if it is safe to move south. If it is then we do. Looking at the short `goSouth` method, do you see any problems? What, in particular, do you think of line 128? It looks like `GameWithMapOnly` has to reach quite deeply into `Location` and know a lot about what `getSouth` will return when there is no such place. This works for the current program but noticing the mixing of different levels of abstraction in the same method can give you some places where fixing up the code will make the next iteration better.

It would be better if we had a `canMoveSouth` method in `Location` so that `goSouth` could just call it to see if it was possible to move. We will put those methods in for the next iteration.

Besides that problem, all we do is get the `Location` from `map` with a call to `findLocation` with the UUID of the location. Then find the UUID of the location to the south, if it is not the “null” UUID, then we can move and we do by updating the `player` (UUID of where the player is) with the southern UUID.

Now, how does `findLocation` search `map` to find the current location? How long does such a search take? And do we care?

13.5 Finding a Match

Sorting, presented in Chapter 10, is an important algorithm to know; you may find yourself needing to sort something in a way that the `Collections.sort` method does not easily support. For example, if your objects do not implement the `Comparable` interface.

Another useful algorithm is *searching*, being able to find a particular object in a collection efficiently. One nice thing about searching is that it gives us some insight into how one determines what *efficient* means. It is possible to write two different versions of a search algorithm and then compare how much work it does. This is known as *algorithm analysis* and is a powerful tool in theoretical computer science, a tool with obvious practical applications. The “analysis” in this section will consist of hand waving and assurances; real analysis requires rigor beyond the scope of the current argument.

Examine Each One: Sequential Search

Imagine that you had a notebook containing every single page of notes you have ever taken; each page is labeled with the date that it was written (we’re imagining here). Unfortunately, someone dropped the notebook, everything fell out, and, in their hurry to get it cleaned up, they jammed in the pages in a random order. All of the pages are in the notebook but in no discernible order. How would you find your notes for the first Monday of last month? How would you even know if you took notes on that date?

The *only* viable approach to the unordered notebook is to look at each and every page, checking the date against the first Monday of last month. Any page you skipped *could* be just the page you were looking for so you cannot skip any. We will grant that there is at most one page for a given day so if you do find the given date you can stop searching.

Because you examine each page in the notebook in sequence, this is known as the *sequential search*. This is the same search used in `findLocation` in `GameWithMapOnly`.

```

84 private Location findLocation(String uuid) {
85     for (int i = 0; i != map.size(); ++i) {
86         if (map.get(i).getUUID().equalsIgnoreCase(uuid)) {
87             return map.get(i);
88         }
89     }
90     return null;
91 }

```

Listing 13.19: `GameWithMapOnly` `findLocation`

This code matches search code we have seen before. The count control variable runs across all the index values for `map`. If a match is found, the matching `Location` is returned. If there is no match, then, after checking every single `Location`, the method returns `null`.

Let’s take a step back and consider how long it takes to search your notebook. Each page takes a certain amount of time to check. That time involves getting to the page, finding the date field, interpreting the date field, and then comparing it. Note that that time depends on who is doing the search. If your dates are in cursive, then the deciphering stage would take much longer for the author’s seven-year old son than it would for you, the author of the notes. So, saying it takes one hour to search for the notes from the first Monday doesn’t really tell us anything. How many notes were there in the book? Who was reading them?

How long does a sequential search of `map` take? Again, it depends on what computer you are running, what Java interpreter, what other programs are running, and how many locations are in `map`. Saying it took one quarter second does not tell us anything.

Computer scientists tend to talk about average times and express time as a function of the size of the data on which the algorithm works. For `map`, the size of the data is the number of entries to be searched. What we are interested in is not how long it takes to find the `Dwarf: Hall: Middle` location but rather how long it takes to find *any* entry (on average) and what happens to that time as the size of `map` varies.

So, how many entries must be checked when we search for a given `Location`? We will make a simplifying assumption: the `map` is completely random. There is no order in the list; thus searching for any given `UUID`

from the front of the list has the same chance of ending on any given entry; also, each entry has the same chance of being searched for. How many entries must be checked if the UUID is not in the list? That is easy: `map.size()`. Let's call that value n . How about for an entry that is in the map? The random nature of the list means that any given entry is found, on average, half way through. For every time we search for the last entry we also search for the first entry. Every long run, past halfway, is averaged together with a short run, less than half way. Rest assured that the number of entries compared, on average, is $n/2$.

What happens if we double the size of map? Whatever amount of time the search took on a given machine will be doubled. And if we multiply the size by 10? Ten times slower. This is known as a *linear* algorithm.

Dismissing Half at a Time: Binary Search

Let's consider a different situation. One author lives in a very small town with an equally small free library. The library is small enough that it would be practical to go through it in an hour, checking each and every book to see if it was *Sorting and Searching* by Donald Knuth. Starting at one end of the stacks and working down to the other, all 1024 books could be checked.

Is that a practical search technique for Knuth's book in your college library? Or, perhaps, in the United States Library of Congress? How *would* you search for the book in a large library? No card catalog permitted.

If the books are randomized you have a task 1000 or 10000 or 100000 times harder than the one in the free library and a sequential search of a long, long time in front of you. Fortunately, very few libraries are randomized. Let's say the books are ordered by author's last name⁷. How would you search the *sorted* library?

Hopefully you would go to the middle of the stacks and look at the author's name there. Seeing it is "Milton", what would you do? You would know that "Knuth" must come before "Milton" so you would dismiss all the books after the middle one you checked. Then you could check the middle of the first half. Seeing "Franklin" you would know that "Knuth" was in the upper half of the lower half. You could dismiss all books before "Franklin". And so on. How long would it take to search for "Knuth" in terms of the number of books?

TK - PICTURE

In the free library, using the smart search algorithm, it would take eleven comparisons at most. If the book were found before the eleventh comparison but after eleven comparisons it would have been found or it is not available. Proof is by the size of the remaining books to check. The sequence of search area sizes is

```
1024
512
256
128
64
32
16
8
4
2
1
```

After checking the last book there is no where else to go. The number of comparisons here is $\lg(n)$, the logarithm of the size of the original search area. If the number of books were doubled, the number of comparisons would go up by 1 rather than double. This search, the *binary search* is faster than the sequential search.

But it requires that the entries be sorted by the field being search for. If the library were ordered on publication date and we only had the author of the book we wanted, it would be no better than a random ordering. To use this algorithm on a list, we need to maintain a sorted list. You will recall that the `Dictionary` class keeps its list in order by `attribute`.

```
140 private int indexOfKey(String matchKey) {
141     int matchingNdx = -1;
```

⁷The Library of Congress number or Dewey Decimal number serve the same purpose. Assume we know Knuth's book's number in the system and all the argument remains valid.

```

142     int lowerNdx = 0;
143     int upperNdx = allEntries.size();
144     // invariant: if matchKey is in allEntries then
145     // matchNdx is on the range [lowerNdx, upperNdx).
146     while (rangeSize(lowerNdx, upperNdx) > 0) { // any remaining entries?
147         int midNdx = (lowerNdx + upperNdx) / 2;
148         int compareMidToMatch = allEntries.get(midNdx).getAttribute()
149             .compareTo(matchKey);
150         if (compareMidToMatch > 0) {
151             // allEntries[midNdx] < matchKey; search upper half of remaining
152             upperNdx = midNdx;
153         } else if (compareMidToMatch == 0) {
154             // allEntries[midNdx] == matchKey; return midNdx
155             matchingNdx = midNdx;
156             break;
157         } else { // if (compareMidToMatch > 0
158             // matchKey < allEntries[midNdx]; search lower half of remaining
159             lowerNdx = midNdx + 1;
160         }
161     }
162     return matchingNdx;
163 }

```

Listing 13.20: Dictionary indexOfKey

The search method uses a helper method, `rangeSize` which just returns the number of entries in the index range `[lowerNdx, upperNdx)`. The range is asymmetric, including `lowerNdx` but excluding `upperNdx`. This is how the indexes for an `ArrayList` are reported if we use `0` and `allEntries.size()`. This is how we have talked about index ranges through out the book.

Lines 141-143 initialize everything. The comment on lines 144-145 talks about an *invariant*. An invariant is a property of a loop which is always true at the top of the loop. That is, the property holds when the loop begins and going all the way through the body of the loop makes sure the property is again true. In the body of the loop it is possible that the invariant is, momentarily, violated but the body of the loop must restore it.

If the invariant on 44-45 is invariant, then our search works. The `if` inside the loop either finds the match (it happens to be at `midNdx`) or it makes the range smaller. Thus it closes in on the matching entry by throwing away half of the entries.

The comments explain where the matching entry must be after the comparison. The `>` and the `<` in the comments are used to make the ordering easier to read; they mean “later in the lexicographic order” and “earlier in the lexicographic order” respectively.

The tricky part of the code is the difference between line 152 and 159. The important thing is that the size of the range being search must get smaller each time through the loop. Since `midNdx` will be between `lowerNdx` and `upperNdx`, it follows that making sure we exclude the entry at `midNdx` makes sure at least one element is removed. Thus line 59 moves `lowerNdx` up *past* `midNdx` since that entry has already been checked. Similarly, line 52 does the same with `upperNdx` but without subtracting 1 because of the asymmetry of the range. By setting `upperNdx` to `midNdx`, `midNdx` is excluded from the range. So long as the range gets smaller each time, eventually we will find the match or the range size will be 0. We are done in either case.

So, why don't we always use binary search? Because the elements must be in sorted order by the search key. When you searched for your notes, the notebook would have had to be in sorted order by date for you to take advantage of binary search. If you did a lot of searches by date, it would be worth sorting it. If, however, you search by date only once in a very long while, it might be worth the cost of sequential search to avoid the task of sorting.

Similarly, if you search by date commonly, then sorting make sense. If you then occasionally search by course name you would not want to have to resort the notebook twice (once into course name order and then back into date order when you perform the more common search).

Because the dictionary is searched with each command and the alias list might grow large, it was determined to sort it and use binary search.

13.6 Summary

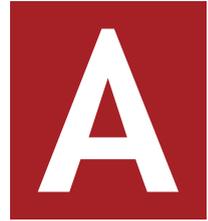
Java Templates

Chapter Review Exercises

Review Exercise 13.1

Programming Problems

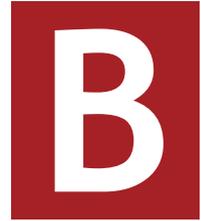
Programming Problem 13.1



Java Language Keywords

The following table lists the reserved words in Java (as of version 6.0); none of these may be used as an identifier in your Java program. Note that while not in the table, `true`, `false`, and `null` are also off limits (they are literals (of the `boolean` and reference types)).

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



References

- [AHD00] *The American Heritage Dictionary of English Language*.
Houghton Mifflin Company, 4 edition, 2000.
Updated 2003.
- [Ben88] Jon Bentley.
More programming pearls: confessions of a coder.
ACM, New York, NY, USA, 1988.
- [Ben00] Jon Bentley.
Programming pearls (2nd ed.).
ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [Bog07] Ian Bogost.
Persuasive Games: The Expressive Power of Videogames.
The MIT Press, Cambridge, Massachusetts, USA, 2007.
- [Con07] Mia Consalvo.
Cheating: Gaining Advantage in Videogames.
The MIT Press, Cambridge, Massachusetts, USA, 2007.
- [Cra84] Chris Crawford.
The Art Of Computer Game Design: Reflections Of A Master Game Designer.
Osborne/McGraw-Hill, New York, New York, USA, 1984.
Electronic version of text <http://www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html>.
- [Dic09] Dictionary.com unabridged (v 1.1).
Mar 2009.
- [DW92] Nell Dale and Chip Weems.
Turbo Pascal.
Houghton Mifflin, Boston, MA, 3 edition, 1992.
- [FFBD04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Dierra.
Head First Design Patterns.
O'Reilly Media, Inc., Sebastapol, CA, USA, 2004.

- [Gee03] James Paul Gee.
What Video Games Have to Teach Us About Learning and Literacy.
Palgrave MacMillan, New York, New York, USA, 2003.
- [Gro07] Agile Manifesto Group.
The agile manifesto.
Technical report, 2007.
<http://www.agilemanifesto.org/principles.html>.
- [Har06] Eliot Rusty Harold.
Java I/O.
O'Reilly Media, Sebastopol, CA, USA, 2nd edition, 2006.
- [Hop47] Smithsonian image 92-13137: Grace hopper's computer bug.
Museum specimen, National Museum of American History, Washington, DC, USA, 1947.
Photo available: <http://americanhistory.si.edu/collections/comphist/objects/bug.htm>;
2009-02-15.
- [HT99] Andrew Hunt and David Thomas.
The Pragmatic Programmer: From Journeyman to Master.
Addison-Wesley Professional, October 1999.
- [Hui55] Johan Huizinga.
Homo Ludens: A Study of the Play Element in Culture.
Beacon Press, Boston, MA, 1955.
- [IMS76] Item 102626678: Gandalf imesai 8800.
Museum specimen, Computer History Museum, Mountain View, CA, USA, 1976.
Photo available: <http://www.computerhistory.org/collections/accession/102626678>; 2009-02-21.
- [Kni00] Reiner Knizia.
Dice Games Properly Explained.
Elliot Right Way Books, London, England, November 2000.
- [KO08] Lawrence Kutner and Cheryl Olson.
Grand Theft Childhood: The Surprising Truth About Violent Video Games.
Simon and Schuster, New York, New York, USA, 2008.
- [KR78] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice Hall, Englewood Cliffs, NJ, February 1978.
- [Ló08] Javier López.
Tetris tutorial in c++ platform independent focused in game logic for beginners.
Web published, December 2008.
CC-Attribution Unported, <http://gametuto.com/tetris-tutorial-in-c-render-independent/>.
- [MAM09] Multiple arcade machine emulator official site.
Retrieved 29 July 2009 from: <http://mamedev.org/>, 2009.
- [New04] James Newman.
Videogames.
Routledge, London, England, 2004.
- [OED71] *The Compact Edition of the Oxford English Dictionary* actually.
Oxford University Press, Oxford, England, 1 edition, 1971.

- [OED89] *Oxford English Dictionary Online*.
Oxford University Press, Oxford, England, 2 edition, 1989.
Accessed online, July 29, 2009.
- [Par99] David Parlett.
The Oxford History of Board Games.
The Oxford University Press, 1999.
- [RBC⁺06] Eric Roberts, Kim Bruce, James H. Cross, II, Robb Cutler, Scott Grissom, Karl Klee, Susan Rodger, Fran Trees, Ian Utting, and Frank Yellin.
The acm java task force: final report.
In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 131–132, New York, NY, USA, 2006. ACM.
- [Sch08] Jesse Schell.
The Art of Game Design: A Book of Lenses.
Morgan Kaufmann, Burlington, MA, 1 edition, 2008.
- [Spi07] Frank Spillers.
What is design? (yes, all 10 definitions!).
http://experiencedynamics.blogspot.com/site_search_usability/2007/10/what-is-design-.html, October 30 2007.
- [SZ04] Kattie Salen and Eric Zimmerman.
Rules of Play: Game Design Fundamentals.
MIT Press, Cambridge, Massachusetts, USA, 2004.
- [Tol54] J. R. R. Tolkien.
The Lord of the Rings.
Houghton Mifflin, 1954.
- [Whi92] Jeff White, editor.
TI INTERNATIONAL USERS NETWORK CONFERENCE. Texas Instruments, September 1992.
Transcript of interview: <http://groups.google.com/group/comp.sys.ti/msg/73e2451bcae4d91a>, 2009-02-02.

Appendix



Java Templates



FANG Color Names

The table on the following page lists all of the color names built-in to the FANG Palette class. The name is given with the “Web” color string specifying the red, green, and blue channels of the color. These names can be used directly in the **static** `getColor` method of `fang.attribute.Palette` or in the **static** `getColor` method of `fang.core.Game`. Note that both `getColor` methods are case-insensitive and they compress out all whitespace within the name string.

Alice Blue	#F0F8FF	Gold	#FFD700	Navajo White	#FFDEAD
Antique White	#FAEBD7	Goldenrod	#DAA520	Navy	#000080
Aqua	#00FFFF	Gray	#808080	Old Lace	#FDF5E6
Aquamarine	#7FFFD4	Grey	#808080	Olive	#808000
Azure	#F0FFFF	Green	#008000	Olive Drab	#6B8E23
Beige	#F5F5DC	Green Yellow	#ADFF2F	Orange	#FFA500
Bisque	#FFE4C4	Honey Dew	#F0FFF0	Orange Red	#FF4500
Black	#000000	Hot Pink	#FF69B4	Orchid	#DA70D6
Blanched Almond	#FFEBCD	Indian Red	#CD5C5C	Pale Goldenrod	#EEE8AA
Blue	#0000FF	Indigo	#4B0082	Pale Green	#98FB98
Blue Violet	#8A2BE2	Ivory	#FFFFFF	Pale Turquoise	#AFEEEE
Brown	#A52A2A	Khaki	#F0E68C	Pale Violet Red	#DB7093
Burlywood	#DEB887	Lavender	#E6E6FA	Papaya Whip	#FFefd5
Cadetblue	#5F9EA0	Lavender Blush	#FFF0F5	Peachpuff	#FFDAB9
Chartreuse	#7FFF00	Lawn Green	#7CFC00	Peru	#CD853F
Chocolate	#D2691E	Lemon Chiffon	#FFFACD	Pink	#FFC0CB
Coral	#FF7F50	Light Blue	#ADD8E6	Plum	#DDA0DD
Cornflower Blue	#6495ED	Light Coral	#F08080	Powder Blue	#B0E0E6
Cornsilk	#FFF8DC	Light Cyan	#E0FFFF	Purple	#800080
Crimson	#DC143C	Light Goldenrod Yellow	#FAD2	Red	#FF0000
Cyan	#00FFFF	Light Green	#90EE90	Rosy Brown	#BC8F8F
Dark Blue	#00008B	Light Grey	#D3D3D3	Royal Blue	#4169E1
Dark Cyan	#008B8B	Light Gray	#D3D3D3	Saddle Brown	#8B4513
Dark Goldenrod	#8B860B	Light Pink	#FFB6C1	Salmon	#FA8072
Dark Gray	#A9A9A9	Light Salmon	#FFA07A	Sandy Brown	#F4A460
Dark Green	#006400	Light Sea Green	#20B2AA	Sea Green	#2E8B57
Dark Khaki	#BDB76B	Light Sky Blue	#87CEFA	Seashell	#FFF5EE
Dark Magenta	#8B008B	Light Slate Gray	#778899	Sienna	#A0522D
Dark Olive Green	#556B2F	Light Slate Grey	#778899	Silver	#C0C0C0
Dark Orange	#FF8C00	Light Steel Blue	#B0C4DE	Sky Blue	#87CEEB
Dark Orchid	#9932CC	Light Yellow	#FFFFE0	Slate Blue	#6A5ACD
Dark Red	#8B0000	Lime	#00FF00	Slate Gray	#708090
Dark Salmon	#E9967A	Lime Green	#32CD32	Slate Grey	#708090
Dark Sea Green	#8FBC8F	Linen	#FAF0E6	Snow	#FFFAFA
Dark Slate Blue	#483D8B	Magenta	#FF00FF	Spring Green	#00FF7F
Dark Slate Gray	#2F4F4F	Maroon	#800000	Steel Blue	#4682B4
Dark Turquoise	#00CED1	Medium Aquamarine	#66CDAA	Tan	#D2B48C
Dark Violet	#9400D3	Medium Blue	#0000CD	Teal	#008080
Deep Pink	#FF1493	Medium Orchid	#BA55D3	Thistle	#D8BFD8
Deep Sky Blue	#00BFFF	Medium Purple	#9370DB	Tomato	#FF6347
Dim Gray	#696969	Medium Sea Green	#3CB371	Turquoise	#40E0D0
Dim Grey	#696969	Medium Slate Blue	#7B68EE	Violet	#EE82EE
Dodger Blue	#1E90FF	Medium Spring Green	#00FA9A	Wheat	#F5DEB3
Fire Brick	#B22222	Medium Turquoise	#48D1CC	White	#FFFFFF
Floral White	#FFFAF0	Medium Violet Red	#C71585	White Smoke	#F5F5F5
Forest Green	#228B22	Midnight Blue	#191970	Yellow	#FFFF00
Fuchsia	#FF00FF	Mint Cream	#F5FFFA	Yellow Green	#9ACD32
Gainsboro	#DCDCDC	Misty Rose	#FFE4E1	FANG Blue	#6464FF
Ghost White	#F8F8FF	Moccasin	#FFE4B5	SCG Red	#A62126

Index

- RectangleSprite, 35
- abstract data type, 142, 293
- abstraction, 13, 143
- actual
 - parameter list, 121, 248
- algorithm, 254
- analog, 20
- animation, 201
- applet, 219
- application, 219
- array, 220
- autoboxing, 258
- autounboxing, 258
- binary, 20
- bit, 21
- block, 91
- byte, 21
- bytecode, 24
- cast operator, 290
- central processing unit, 11, 21
- classpath, 28
- collection, 155, 167
- comment, 63
- compiler, 24
- computer
 - game, 12
- computer bug
 - first, 26
- computer program, 10
- constructor, 18, 80
 - default, 129
 - super**, 128
- container, *see* clllection167
- CPU, *see* central processing unit
- decorator, 325
- decorator pattern, *see* decorator
- default constructor, 209
- delegation, 13
- digital, 20
- directory, *see* folder
- disk drive, 21, 232, 272
- Do not Repeat Yourself Principle, 48, 282
- DRY, *see* Do not Repeat Yourself Principle, 191
- DRY Principle, *see* Do not Repeat Yourself Principle
- dynamic attribute, 248
- dynamic properties, 168
- EBNF, *see* Extended Backus-Naur Form
- EmptyGame.java, 28
- eof-controlled loop, 235, 236
- equals, 151
- exception, 236
- expression, 94, 95
- Extended Backus-Naur Form, v, 26
- extension point, 59
- factory, 275
- FANG, *see* Freely Available Networked Game Engine
- fetch-execute-store loop, 11
- field, 62, 87
- file, 233
- file system, 213, 232
- folder, 233
- formal
 - parameter list, 121, 248
- Freely Available Network Game Engine, 25
- Freely Available Networked Game Engine, 17
- Freely Available Networked Game Engine, 30
- Game
 - methods
 - randomDouble, 64
- game
 - component, 4, 86
 - active, 5
 - passive, 5
 - computer, *see* computer game
 - definition, 4
 - game design
 - FluSimulator, 156
 - Pig, 245
 - game design

- EasyDice*, 73
- Hangman*, 213
- NewtonsApple*, 58
- RescueMission*, 187
- SoloPong*, 113
- TicTacToe*, 139
- game loop, 7, 32
- game mod, 278
- getter, 105
- Hopper, Admiral Grace Murray, 26
- I/O, *see* input/output devices
- image
 - pixel-based, 42
 - vector-based, 42
- IMSAI 8800, 23
- input/output devices, 21
- interface**, 180
- interpreter, 24
- is a, *see* is-a
- is-a, 190, 296
- iteration, 13
- Java, 24
- java, 29
- javac, 28
- javadoc, 102
- level editor, 278
- lexicographic order, 232
- list
 - literal, 257
- loop
 - count-controlled, 223
 - nested, 193
 - sentinel-controlled, 224
- loop-control variable, 162
- main method, 218
- method, 63, 87, 119
 - main, *see* main method
 - name, 63
 - parameter list, 63
 - return type, 63
 - signature, 63
 - visibility, 63
- moderator, 6
- multiple inheritance, 294
- multitasking, 12
- naming conventions, 93
 - method, 94
 - type, 94
- variable, 93
- need to know principle, 246
- nested classes, 295
- new**, 89
- null**, 68
- null character, 216
- Object, 161
- open source software, 31
- operating system, 218
- operating system, 24
- OvalSprite, 38
- overload, 98
- override, 60, 82
- parallel, 12
- parameter, 129
- path name, 233
- pixel, 34
- problem solving techniques, 13
- processor bound, 12
- public interface, 75, 87
- RAM, *see* random access memory
- random access memory, 272
- random access memory, 10, 21, 232
- reference, 88
- rule follower, 7
- scope, 91, 129
- scope hole, 129
- score, 59
- screen coordinates, 34
- selection, 13
- semantics, 162
- sequence, 13
- sequential, 12
- setter, 105
- short-circuit Boolean evaluation, 201
- signature
 - method, *see* method signature, 119
- single inheritance, 294
- software engineering, 293
- sprite, 34
- static**, 143
- static attribute, 248
- static final**, 93
- static properties, 168
- strategy, 10
- stub method, 217
- text adventure game, 317
- thread of control, 12
- token, 250, 320

traverse, 199

tree, 270

type, 86, 87

variable, 89

- declaration, 89

- local, 91

video game loop, 59, 141, 192

von Neumann, John, 23, 81

- architecture, 23, 81