The cs111 Text

Jennifer Stephan and Allen Downey

First Edition

The cs111 Text

First Edition

This book was typeset by the author using LaTeX and dvips, which are both free, open-source programs.

Acknowledgements

The material presented in this book has been developed collaboratively by professors at Wellesley College. The microworlds—including BuggleWorld, PictureWorld and TurtleWorld—and the Java Execution Model (JEM) were developed by Franklyn Turbak.

Other professors who have added to and refined this material include Jennifer Stephan, Constance Royden, Elaine Yang, LeeAnn Tzeng, Jean Herbst, and Sergio Alvarez.

Parts of the text are extracted from *How To Think Like a Computer Scientist*, by Allen Downey.

Chava Kronenberg is a Wellesley student that worked for a summer with these materials, creating the original source files based on Jennifer Stephen's lecture notes.

Contents

Chapter 1

The way of the program

The goal of this book, and this class, is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. By that I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

On one level, you will be learning to program, which is a useful skill by itself. On another level you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 What is a programming language?

The programming language you will be learning is Java, which is relatively new (Sun released the first version in May, 1995). Java is an example of a **high-level language**; other high-level languages you might have heard of are Pascal, C, C++ and FORTRAN.

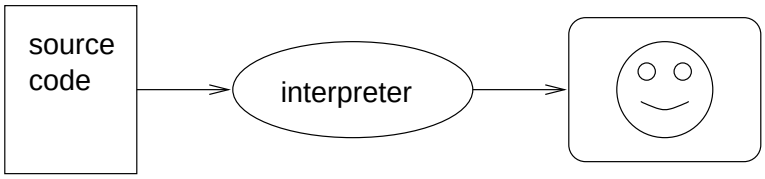
As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is *much* easier to program in a high-level language; by "easier" I mean that the program takes less time to

write, it's shorter and easier to read, and it's more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few special applications.

There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.



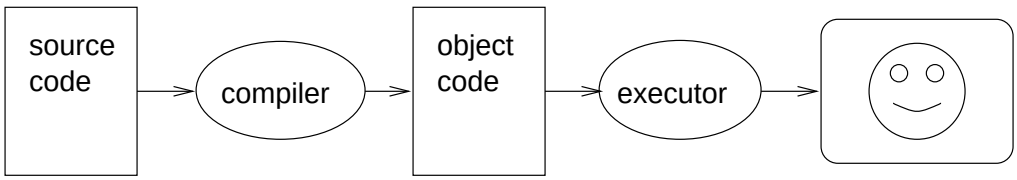
The interpreter reads the source code...

... and the result appears on the screen.

A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named `program.c`, where "program" is an arbitrary name you make up, and the suffix `.c` is a convention that indicates that the file contains C source code.

Then, depending on what your programming environment is like, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named `program.o` to contain the object code, or `program.exe` to contain the executable.



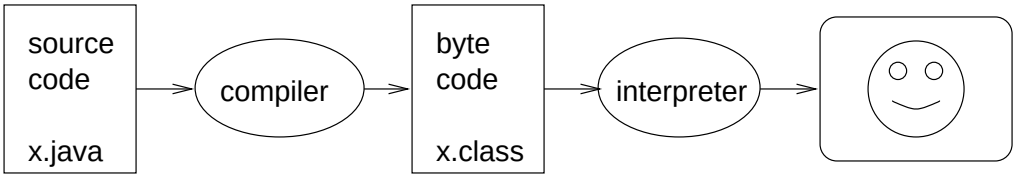
The compiler reads the source code...

... and generates object code.

You execute the program (one way or another)...

... and the result appears on the screen.

The Java language is unusual because it is both compiled and interpreted. Instead of translating Java programs into machine language, the Java compiler generates Java byte code. Byte code is easy (and fast) to interpret, like machine language, but it is also portable, like a high-level language. Thus, it is possible to compile a Java program on one machine, transfer the byte code to another machine over a network, and then interpret the byte code on the other machine. This ability is one of the advantages of Java over many other high-level languages.



The compiler reads the source code...

... and generates Java byte code.

A Java interpreter reads the byte code...

... and the result appears on the screen.

Although this process may seem complicated, the good news is that in most programming environments (sometimes called development environments), these steps are automated for you. Usually you will only have to write a program and type a single command to compile and run it. On the other hand, it is useful to know what the steps are that are happening in the background, so that if something goes wrong you can figure out what it is.

1.2 What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The instructions (or commands, or statements) look different in different programming languages, but there are a few basic functions that appear in just about every language:

input: Get data from the keyboard, or a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

testing: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of functions that look more or less like these. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple functions.

1.3 Three big ideas

There are three important concepts at the core of this class: Divide, Conquer and Glue, Abstraction, and Modularity.

1.3.1 Big Idea 1: Divide, Conquer and Glue

Divide, conquer and glue is the primary technique we will use to solve programming problems. Suppose that you have a problem P and you want its solution S. Divide, conquer and glue works in the following way:

Divide the problem into sub-problems.

Conquer by solving the individual sub-problems.

Glue the sub-solutions into one big solution!

We use this type of problem-solving every day. Let's say you have a research paper due for English. You might divide that problem by first doing the research, then taking notes from different sources on notecards, then sitting down at a computer to write the paper. In order to 'conquer' the first part, perhaps you would need to go to the library or research online. The second part might require buying notecards and writing down all the notes, and you might conquer the third portion by going to the LTC for the evening. In any case, you have conquered all the sub-problems, and after gluing them all together you would have a completed English paper!

1.3.2 Big Idea 2: Abstraction

Abstraction is a way of capturing common patterns by taking two or more things that look the same from a certain perspective and giving them a common name. Then, you can just reference the name to reference either item.

Let's take the two cars in my garage. One is a red 1993 Toyota, the other is a white 1994 Toyota. I can substitute one for the other, and they have many similar qualities. They have the same "shape". Are they identical? No. Maybe one has a different engine than the other, maybe one has different tires. Do I care that they aren't identical? No, because I can use either car without knowing all the minute differences between the two. I can call both of these Toyotas a car.

The power of abstraction is that we can use components without knowing all the details of how they work. That makes it possible to build large systems (or programs) without being overwhelmed by complexity.

1.3.3 Big Idea 3: Modularity

Finally, we have modularity, which is a means of connecting the layers of abstraction. Large systems are built up from components called **modules**. The interfaces between modules are designed so they can be put together in a mix and match way, like Legos. For instance, pieces of existing code can be used as building blocks for much larger, complex pieces of code.

1.4 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.4.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in Java than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.4.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program. In Java, run-time errors occur when the interpreter is running the byte code and something goes wrong.

The good news for now is that Java tends to be a **safe** language, which means that run-time errors are rare, especially for the simple sorts of programs we will be writing for the next few weeks.

Later on in the semester, you will probably start to see more run-time errors, especially when we start talking about objects and references (Chapter 8).

In Java, run-time errors are called **exceptions**, and in most environments they appear as windows or dialog boxes that contain information about what happened and what the program was doing when it happened. This information is useful for debugging.

1.4.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.4.4 Experimental debugging

One of the most important skills you will acquire in this class is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. Conan Doyle's *The Sign of Four*).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from *The Linux Users' Guide* Beta Version 1).

In later chapters I will make more suggestions about debugging and other programming practices.

1.5 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3+3 = 6$ is a syntactically correct mathematical statement, but $3 = +6\$$ is not. Also, H_2O is a syntactically correct chemical name, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as I know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax and semantics—there are many differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.6 Glossary

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like Java that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute. Also called "machine language" or "assembly language."

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language: Any of the languages people speak that have evolved naturally.

portability: A property of a program that can run on more than one kind of computer.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to be executed.

byte code: A special kind of object code used for Java programs. Byte code is similar to a low-level language, but it is portable, like a high-level language.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

parse: To examine a program and analyze the syntactic structure.

token: One of the basic elements, like a word or symbol, that make up the syntactic structure of a program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

exception: An error in a program that makes it fail at run-time. Also called a run-time error.

logical error: An error in a program that makes it do something other than what the programmer intended.

debugging: The process of finding and removing any of the three kinds of errors.

Chapter 2

Object Oriented Programming

2.1 Buggles

Java is an object-oriented language, which means that programs construct and manipulate "objects" inside the computer that (usually) represent objects in the real world. More specifically, the objects perform computations by passing messages to each other.

To start learning about Java and object oriented programming, we will use Buggles. Buggles are Java objects that live in Buggleworld, which is another Java object. Buggles are represented on the screen as icoscoles triangles; Buggleworld is represented as a grid of squares. Buggles can move around in BuggleWorld, they can paint in BuggleWorld, they can change direction, and they can drop and eat bagels.

Buggles and BuggleWorld are not part of the Java language. They are part of a program written by Franklyn Turbak based on Turtle graphics, part of the Logo programming language.

2.2 A Buggle is born

Here is the first line of Java code we will see. It creates a new Buggle using the `new` operator:

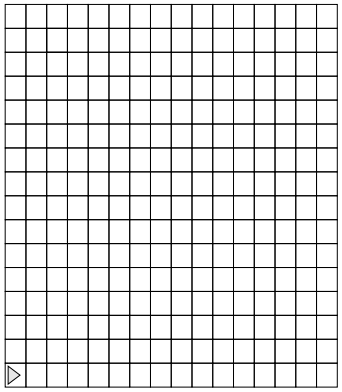
```
Buggle becky = new Buggle();
```

A line of code like this that performs an action is called a **statement**. This one is an **assignment** statement because it assigns a value to the name `becky`.

The left side of the assignment is a **declaration**. It declares that the name `becky` will refer to an object of type Buggle. The right side of the assignment

creates the new object. Once this statement completes, we can use the name `becky` to refer to the new object.

In lab you will learn how to put this statement into a Java program, compile it, and execute it. When you do, you will see a graphical representation of BuggleWorld with a red icosceles triangle in the lower left corner. That's `becky`. She looks a little like this:



Every Buggle in BuggleWorld starts out looking like this, but you can change their properties by sending them messages. The properties are called **instance variables** (in some Java documentation they are also called **fields**). For Buggles, the instance variables are position, heading, color and brush state.

position: The location of the Buggle in BuggleWorld, specified in coordinates on an (x,y) axis. New Buggles are initially in the lower left corner, with coordinates (1,1). Coordinates increase to the right along the x axis and up along the y axis.

heading: The compass direction the Buggle is facing, which is always one of the values NORTH, SOUTH, EAST and WEST. Buggles are born facing EAST.

color: What color the Buggle is, and what color trail it leaves. Buggles are born red.

brushDown: Whether or not the Buggle's brush is down (in the painting position). When the value of this variable is `true` the Buggle leaves a colored trail. When `false` it doesn't. Buggles are born with their brushes down.

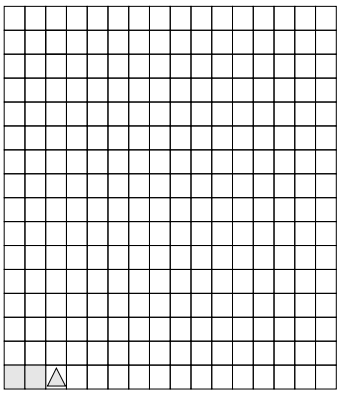
2.3 Changing instance variables

You can change the state of a Buggle by sending it a message. The messages you can send a Buggle include `forward`, `backward`, `left`, and `right`. The following statements move `becky` around.

```
becky.forward();
becky.forward();
becky.left();
```

These statements are called **method invocations**. When they execute, they send messages to `becky`, in this case the messages `forward`, `forward` and `left`. `becky` executes these instructions one at a time, in order.

The result is that the graphical representation of `becky` moves and her instance variables change. Her position is now (3,1) and her heading is now NORTH.



Notice that this movement also has a side-effect. Since `becky`'s brush was down when she moved, she left a stripe of (red) paint in her path.

Other Buggle methods include `brushUp`, `brushDown`, `setPosition`, and `setHeading`.

2.4 Methods with arguments

Some methods require additional information when they are invoked, information that controls *how* the action is performed.

For example, when you invoke the `setColor` method, you have to provide a Color object:

```
becky.setColor (Color.blue);
```

The additional information you provide is called an **argument**. In this case the argument is one of the built-in colors, `Color.blue`. This has the effect of making `becky` blue. The next time she moves, she will leave a blue trail.

Other built-in colors include

```
black     blue    cyan   darkGray   gray   lightGray
magenta   orange  pink   red        white  yellow
```

There is an alternate form of the methods `forward` and `backward` that takes an integer as an argument. For example,

```
becky.forward (5);
```

Moves `becky` forward five spaces (and paints a stripe along the way, if her brush is down). As an exercise, draw what you think BuggleWorld will look like after these statements execute, and then try it out.

2.5 Variable declaration

A **variable** is a name that refers to an object (like a Buggle) or a value (like the number 3). A variable declaration creates a new variable and declares what **type** it is. So far the only types we have seen are `Buggle` and `Color`, but there are other built-in types, like `int` and `Point`, and other types in BuggleWorld, like `Direction`.

The simplest form of a declaration looks like this:

```
Type name;
```

where `Type` is the type of the variable and `name` is the name. The semi-colon at the end is mandatory, like the period at the end of a sentence

A variable declaration all by itself doesn't assign a value to the new variable, so it is very common to combine it with an assignment, like this:

```
Type name = value;
```

Here are a few more examples:

```
int n = 3;
Color myFavorite = Color.green;
```

By convention, variable names always begin with lowercase letters. If a variable name contains multiple words, the first letter of each word (except the first) is capitalized. Variable names can be any length, and may also include numbers (but not first).

2.6 Classes

A **class** is a category. For example, the Buggle class is the category that contains all Buggle objects. The Color class, you will not be surprised to hear, contains all the Color objects.

Conversely, every object belongs to a class, which is why objects are also called **instances**—each one is a specific instance of its general class.

Each class specifies the set of methods that can be applied to the objects in that class. The methods we have seen that apply to Buggles are all defined in the Buggle class, which is in a file named `Buggle.java`.

For every class there is a corresponding type, so when you create Buggle objects you assign them to variables with type Buggle. When you create Color objects, you will not be surprised to hear, you assign them to variables with type Color.

2.7 Constants

Values like `3` and `Color.green` are **constants**, basic values that never change. Like variables and objects, constants also have types. Numbers like `3` have the type `int`, short for integer. `Color.green` has type `Color`.

In order to assign a constant to a variable, the constant and the variable have to have the same type. So in the previous examples, we can assign `3` to an `int` and `Color.green` to a `Color`, but not the other way around.

Some constants are associated with a class, as the color `green` is associated with the `Color` class. To use those constants, you have to specify both the name of the class and the name of the constant.

As another example, the `Direction` class contains the constants `EAST`, `WEST`, `NORTH` and `SOUTH`. You can pass them as arguments to `setHeading`:

```
becky.setHeading (Direction.EAST);
```

This statement tells `becky` to face east.

Once you assign a value to a variable, you can use the variable anywhere you would use the value. For example:

```
int n = 3;
becky.forward (n);
Direction d = Direction.EAST;
becky.setDirection (d);
```

2.8 Expressions

An **expression** is a collection of variables and constants that can be evaluated to yield a value. You are probably familiar with mathematical expressions like `7 + 3`, which has the value 10. You have probably also seen expressions with variables, like `7 + n`, which also has the value 10, if the value of `n` happens to be 3.

Arithmetic symbols like `+` are called **operators** because they perform mathematical operations. Java provides operators for addition `+`, subtraction `-`, multiplication `*` and division `/`. Division is sometimes confusing because when you divide integers, Java performs integer division, which always rounds down to the nearest integer. So `13/4` is 3, and `99/100` is 0.

It is important to remember the difference between an expression and a statement. A statement is a Java command that does something. So far we have seen three kinds of statement: declarations, assignments and method invocations.

An expression can appear on the right-hand side of an assignment, and it can appear as an argument for a method invocation, but an expression all by itself is not a statement. For example, `n + 5;` is not a legal statement all by itself.

Also, the left side of an assignment has to be the name of a variable. It can't be an expression. So

```
n + 5 = 7;
```

is not legal. This example demonstrates something very important:

> An assignment statement is not a statement of equality.

It is an (unfortunate) coincidence that the assignment operator, `=`, looks a lot like the equality symbol used in mathematics. But it is not the same thing! The best way to think about an assignment is "a statement that evaluates the expression on the right and assigns the result to the variable on the left."

2.9 Constructors

For every class there is a constructor that creates new instances of that class. To invoke the constructor, use the `new` operator and the name of the class. We have already seen one example:

```
Buggle becky = new Buggle();
```

Here's how to create a `Point` object that represents the location $(4, 2)$:

```
Point p = new Point (4, 2);
```

Notice that the names of classes begin with upper-case letters. The types that begin with lower-case letters are not classes, and they obey some different syntax rules. For example, you cannot use the `new` command to create an `int` object.

You can use a `Point` object as an argument to the `setPosition` method.

```
becky.setPosition (p);
```

This invocation causes `becky` to leap to the position $(4, 2)$ from anywhere in BuggleWorld.

2.10 Sending messages

When you invoke a method on an object, you send a message to the object telling it what to do. The general form of a method invocation is

```
object.method (arguments);
```

where `object` is the name of the object, `method` is the name of the method, and `arguments` is a list of arguments that can be arbitrarily long, or empty. We have already seen an example of an empty argument list:

```
becky.forward ();
```

And a "list" with a single argument:

```
becky.forward (5);
```

When a method takes more than one argument, the arguments are separated by commas. For example, one way to create a new `Color` object is to specify how much red, green, and blue light to mix, on a scale of 0–255:

```
Color purple = new Color (150, 0, 210);
```

In general, the arguments you provide have to match the arguments the method is expecting to receive. For example, if you invoke `forward` like this:

```
becky.forward (1, 2);
```

The compiler will complain, because it was expecting either one integer argument or no arguments; two arguments is illegal.

2.11 Void and fruitful methods

The methods we have seen so far make Buggles do things, and we can see the results in BuggleWorld. But invoking these methods does not produce a value as a result.

Methods that produce values are called **fruitful**; methods that don't are called **void**.

Invoking a void method is a statement all by itself, but invoking a fruitful method is just an expression. You cannot use a void method as an expression, so

```
int n = becky.forward ();
```

is illegal. If you try it, you will get a compile-time error.

If you use a fruitful method invocation as a statement, you won't get an error (unfortunately), but the statement doesn't do anything. For example,

```
becky.getColor ();
```

gets `becky`'s color, but it doesn't do anything with the result! It is common (and much more useful) to put fruitful method invocations on the right-hand side of an assignment.

```
Color c = becky.getColor ();
```

This statement takes the result from `getColor` and assigns it to `c`. Then we can use `c` as an argument to another method:

```
bobby.setColor (c);
```

The result is that `bobby` is transformed to whatever color `becky` was.

In this case, we call `c` a **temporary variable** because we used it to hold a value temporarily while we passed it from one object to another.

It is also possible to cut out the middleman:

```
bobby.setColor (becky.getColor ());
```

This is an example of a **nested** method invocation, since `getColor` is nested inside `setColor`. Statements like this are evaluated from the inside (innermost parentheses) out. So, we invoke `getColor` first and then pass the result as an argument to `setColor`. The effect is the same as the longer version.

2.12 Comments

This is the first piece of code we have seen that is complicated enough to deserve a comment. A **comment** is a line of English that you can include in a program to explain what the program does or how it works. Comments don't affect what the program does; they just make it easier to read:

```
// give bobby becky's color
bobby.setColor (becky.getColor ());
```

Comments begin with two forward slashes (`//`, not `\\`) and go until the end of the line.

```
// comments usually get a line to themselves
// but occasionally you might want to tack one onto
bobby.setColor (becky.getColor ());   // the end of a statement
```

2.13 More nesting

I've said this before, but it's so important I'm going to say it again:

> Invoking a void method is a statement all by itself, but invoking a fruitful method is just an expression. You have to do something with the result.

As we have already seen, you can use a fruitful method invocation anywhere you can use an expression: on the right side of an assignment, or as an argument. You can also invoke a method on the result of a previous invocation:

```
Color c = bobby.getColor().darker();
```

The `darker` method belongs to the Color class. If you invoke it on a Color object, it returns a new Color object that is a darker version of the original. Note that the original Color does not change!

So this statement gets `bobby`'s current color and creates a new color that is a little darker. The result is assigned to `c`. This statement has no effect on `bobby`.

If we want to change `bobby`'s color we have to invoke `setColor`:

```
bobby.setColor (c);
```

Again, you can eliminate the temporary variable and make the statement even more complicated!

```
bobby.setColor (bobby.getColor().darker());
```

If you read this statement from left to right, you are likely to be confused. It is better to read it in the order it will be evaluated, from the inside out.

2.14 Contracts

Every class has a contract that specifies what methods it implements and what constants it provides.

For example, the `Buggle` contract contains a list of methods you can invoke on Buggles, including two forms of `forward`:

> `public void forward ()`
>
> Moves this buggle forward one step (in the direction of its current heading). Complains if the buggle is facing a wall.
>
> `public void forward (int n)`
>
> Moves this buggle forward n steps. If the buggle encounters a wall along the way, it will stop and complain.

Just to make things confusing, the way methods are presented in the contract is different from the way you invoke them. But you can read the contracts the same way you read code.

The first line says that `forward` is a public void method that takes no arguments. `public` means that you can invoke this method from other classes, as opposed to `private` methods, which can only be invoked from other Buggle methods. `void` means that this method does not return a value.

The second form of `forward` is similar except that it takes a single argument, named `n`, that has type `int`. It is illegal to invoke `forward` with any kind of argument other than `int` or with more than one argument.

By the way, it might be entertaining to know that the argument is called `n`, and it is convenient for the documentation to refer to it ("n steps"), but when you invoke the method, you don't have to worry about the name. You can pass any integer expression as an argument:

```
becky.forward (5);
int numberOfSteps = 13;
becky.forward (numberOfSteps);
becky.forward (numberOfSteps + 5);
```

These are all legal. The argument in the first invocation is an integer constant, the second is a variable with type `int` and the third is an expression that, when evaluated, yields an integer.

Here is the documentation for `getColor`:

`public Color getColor ()`

Returns the color of this buggle.

Instead of the word void, we see the word Color before the name of the method. That means that this is a fruitful method that returns an object of type Color. It takes no arguments.

All of the methods in the Buggle class are **instance methods**, which means that you have to invoke them on an instance of the class, i.e. a Buggle. Later we will see **class methods**, which don't require an instance.

2.15 Glossary

statement: A line of a Java program. The three kinds of statement we have seen are declarations, assignments and method invocations.

declaration: A statement that creates a new variable and declares its type.

assignment: A statement that gives a value to a variable.

method invocation: A statement that invokes a method, which has the effect of sending a message to an object.

object: A value in a program that can recieve messages and perform actions.

instance: Another name for an object, the term "instance" emphasizes the notion that all objects are an instance of a category.

class: A category of objects. A class defines the set of operations (methods) that can be invoked on its instances.

constructor: A special method that creates new objects. It is invoked using the `new` operator, not the usual method invocation syntax.

operator: A basic operation, including arithmetic operators as well as `new` and the assignment operator, `=`.

variable: A named location that can contain a value.

value: One of the basic units programs manipulate and compute.

type: Variables, values and objects all belong to types. For every class, there is a corresponding type.

instance variable: A variable inside an object that contains information about the state of the object.

temporary variable: A variable used to hold a value temporarily.

argument: An expression that you provide when you invoke a method. The value of the argument is passed to the method.

return value: The result of invoking a fruitful method.

method: One of the operations an object can perform, or one of the messages that can be sent to an object.

void method: A method that does not have a return value. Invoking a void method is a statement.

fruitful method: A method that has a return value. Invoking a fruitful method is an expression.

instance method: A method that is invoked on an object, which has the effect of passing a message to the object.

expression: A collection of variables, values, operators and method invocations that can be evaluated to yield a value.

comment: A line of description, in English, that is included in a program to explain how it works.

Chapter 3

Methods

3.1 Debugging

There are two stages of debugging: getting the program to compile and run, and getting the program to do the right thing. The first stage is mostly a problem with syntax, and the hardest part is usually figuring out what the error messages mean. The second stage is a problem with semantics; the program you wrote does not mean what you wanted it to mean.

At this point, the first step is to figure out what the program is doing. A useful command for that is `System.out.println`. As debugging tools go, it's a little clumsy, but it can be a big help.

In the simplest form, the **print statement** looks like this:

```
System.out.println ("Hello");
```

Actually, calling it a print statement is misleading, since it doesn't print anything on paper. What it does, in most environments, is pop up a window, called a **console**, where it displays whatever you tell it to display. In this case, we told it to display the word `Hello`.

The quotation marks are necessary to indicate that this is a `String`, which is yet another Java type. If you leave out the quotes, Java thinks `Hello` is the name of a variable, and it complains if there isn't a variable by that name.

On the other hand, if a variable does exist, `println` displays its current value:

```
int x = 5;
System.out.println (x);
```

The output of this code is `5`.

While you are debugging your code, you can add print statements almost anywhere. When you are done, you might want to remove them, to avoid clutter. If you think you might need them later, you can comment them out.

```
int x = 5;
//System.out.println (x);
```

The phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

`println` is short for "print line," because after each line it adds a special character, called a **newline**, that causes the cursor to move to the next line of the console. The next time `println` is invoked, the new text appears on the next line.

Often it is useful to display the output from multiple print statements all on one line. You can do this with the `print` command:

```
System.out.print ("Goodbye, ");
System.out.println ("cruel world!");
```

In this case the output appears on a single line as `Goodbye, cruel world!`. Notice that there is a space between the word "Goodbye" and the second quotation mark. This space appears in the output, so it affects the behavior of the program.

3.2 Concatenation

Earlier I said that the mathematical operators only work with numbers, but that is not completely true. The `+` operator works with `String`s, although it does not do exactly what you might expect.

For `String`s, the `+` operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. So `"Hello, " + "world."` yields the string `"Hello, world."` and `fred + "ism"` adds the suffix *ism* to the end of whatever `fred` is, which is often handy for naming new forms of bigotry.

Concatenation is handy for combining Strings and other values inside a print statement:

```
int bananas = 17;
System.out.println ("Yes, the number of bananas is " + bananas);
```

You might wonder how Java deals with an expression like `"Yes, the number of bananas is " + bananas`, since one of the operands is a `String` and the other is a `int`. Well, in this case Java is smart on our behalf; it automatically converts the `int` to a `String` before it concatenates.

This kind of feature is an example of a common problem in designing a programming language, which is that there is a conflict between **formalism**, which is the requirement that formal languages should have simple rules with few exceptions, and **convenience**, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers (who are spared from rigorous but unwieldy formalism), but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

Nevertheless, it is handy to know that whenever you try to "add" two expressions, if one of them is a `String`, then Java will convert the other to a `String` and then perform string concatenation. What do you think happens if you perform an operation between an integer and a floating-point value?

3.3 BuggleWorld

While we have been playing with Buggles we haven't done much with BuggleWorlds. The BuggleWorld class defines the methods that BuggleWorlds can execute.

One of these methods is the constructor. You can create a new BuggleWorld in the usual way:

```
BuggleWorld world = new BuggleWorld ();
```

In the programs you have used so far you have not had to create your own BuggleWorlds; we gave you code to do that.

The only other method BuggleWorlds can execute is `run`. Try executing it, either by invoking it directly, or by using the GUI (graphical user interface). In its current form, it doesn't do much. In fact, it does nothing.

To fix that, we have to **extend** the existing BuggleWorld class and create a new class, which we will call `MyBuggleWorld`. The new class is just like the old class except that it **overrides** the old `run` method and replaces it with something more interesting. Here's how it looks in Java.

```
public class MyBuggleWorld extends BuggleWorld{

    public void run() {
        Buggle becky = new Buggle();
        becky.setPosition (new Point (4,2));
    }
}
```

The first line says that we are creating a new class named `MyBuggleWorld` that is based on the existing class `BuggleWorld`. The second line says that we are defining a new method, named `run`, that the new class will be able to execute. It is a void method that takes no arguments.

The code inside `run` is nothing new. It creates a Buggle and moves it to the point $(4, 2)$.

3.4 The Java Execution Model

The **Java Execution Model** (JEM) is a way to visualize what is going on when a piece of code executes. It is useful for debugging and it provides a visual aid to otherwise abstract concepts. If you are ever in doubt about what a piece of code does, take out a piece of paper and draw out the JEM.

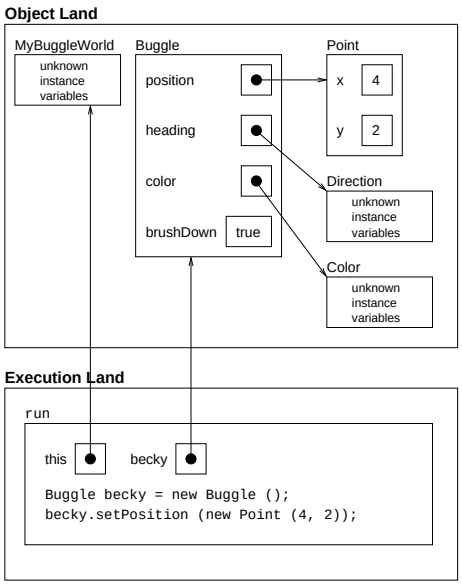
The JEM is divided into two regions: **Object Land**, where objects live, and **Execution Land**, where the code is executed.

3.5 Execution Land

Execution Land displays the evolution of a piece of code as it executes. It is inhabited by execution frames, one for each method that is currently running.

Every execution frame has two parts, the set of variables used in the method, and the sequence of statements that make up the body of the method.

The following figure shows a JEM with a single execution frame. The name of the method, `run`, is outside the frame. Inside the frame are variables named `this` and `becky`. The code is the body of `run`.



The variable named `this` is a special variable that is created by Java; you don't have to declare it. When you send a message to an object, the object uses `this` to refer to itself. In this example, `this` refers to the MyBuggleWorld that received the `run` message.

3.6 Object Land

In Object Land, each object is represented as a group of instance variables that describe the object at a specific point in time.

This JEM shows five objects: a MyBuggleWorld object, a Buggle, a Point, a Color and a Direction. Each is labelled with its type.

For some objects we know what the instance variables are and we can show them in the JEM. For other objects the internal state is opaque.

There are two kinds of value a variable might hold, primitive values and object references. Integers are primitive types, and so are the truth values `true` and `false`. When a variable contains a primitive value, we write the value inside a box and write the name of the variable outside. The instance variables `brushDown`, `x` and `y` are examples.

Object references are, um, references to objects. In the JEM they are represented by a dot and an arrow pointing to an object.

3.7 Writing your own methods

Take a look at the following code and figure out what it does. As an exercise, draw a picture of what BuggleWorld looks like after the program runs.

```
public class MyBuggleWorld extends BuggleWorld{

  public void run() {
    Buggle becky = new Buggle ();
    becky.setPosition (new Point (4,2));
    Buggle bobby = new Buggle ();
    bobby.setColor (Color.blue);

    // becky draws a box 3 steps on each side
    becky.forward (2);
    becky.left ();
    becky.forward (2);
    becky.left ();
    becky.forward (2);
    becky.left ();
    becky.forward (2);
    becky.left ();

    // bobby draws a box 3 steps on each side
    bobby.forward (2);
    bobby.left ();
    bobby.forward (2);
    bobby.left ();
    bobby.forward (2);
    bobby.left ();
```

```
    bobby.forward (2);
    bobby.left ();
  }
}
```

Notice that `becky` and `bobby` execute pretty much the same code. Wouldn't it be more efficient to capture this pattern so that we could simply tell the Buggles to draw a box? Of course!

To do that, we have to create a new kind of Buggle that can execute a new method, called `box`. We can extend Buggles the same way we extended BuggleWorld:

```
class Box2Buggle extends Buggle {

   public void box () {
     this.forward (2);
     this.left ();
     this.forward (2);
     this.left ();
     this.forward (2);
     this.left ();
     this.forward (2);
     this.left ();
   }
}
```

The new class is called `Box2Buggle`. A Box2Buggle object can do everything a Buggle can, but it can also execute the `box` method, which regular old Buggles can't.

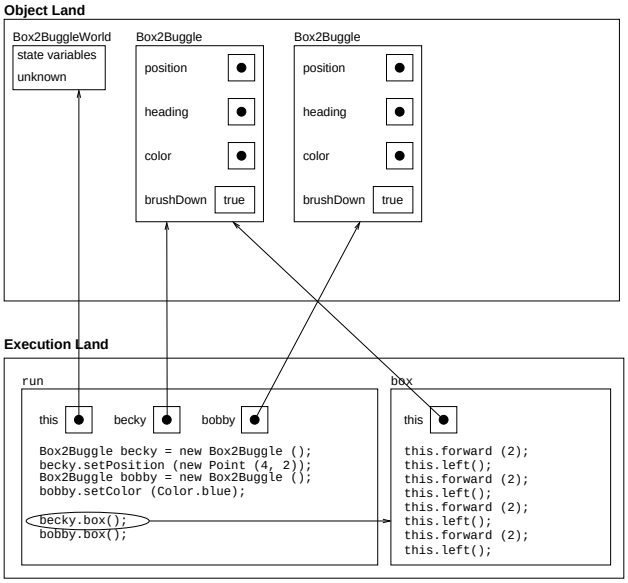
When we create `becky` and `bobby` we have to use the constructor for the `Box2Buggle` class, and assign the result to a variable with type `Box2Buggle`:

```
public class Box2World extends BuggleWorld {

    public void run () {
       Box2Buggle becky = new Box2Buggle ();
       becky.setPosition (new Point (4,2));
       Box2Buggle bobby = new Box2Buggle ();
       bobby.setColor (Color.blue);

       becky.box ();
       bobby.box ();
    }
}
```

Here is what the JEM looks like when we invoke `box` on `becky`:



For the object references inside the Buggle objects, I just drew dots and omitted the objects they refer to.

Notice that there are now two execution frames, one for `run` and one for `box`. Inside `run` I added a dot that shows which line of the program is currently excecuting. In this JEM, we are in the middle of executing `box`.

The arrow points from the method invocation to the execution frame of the method. Inside `box`, there is only one variable, `this`. As always, `this` refers to the object that received the message, which is `becky`.

This is the first case we have seen where two variables refer to the same object. This situation is called **aliasing** because `becky` is going by an alias. Inside `run`, she is known as `becky`, but inside `box` she goes by the nickname `this`. Of course, anything we do to `this` is going to affect `becky`.

This example demonstrates one of the benefits of methods. By writing a box() method, we avoided writing the same code twice. Imagine how much code we would save if there were 10 Buggles making boxes! Methods also make code more readable, less error prone, and easier to reuse.

3.8 Writing methods with parameters

In the previous example we wrote a method, `box`, that made a Box2Buggle draw a box 3 units on a side. But, what if we wanted the Buggle to draw a box with some other size, say 6? Would we have to write a whole new method? Not if we use **parameters**. Parameters are variables that store the arguments you provide when you invoke a method.

Here is a version of `box` that can take the size of the box as a parameter.

```
class Box2Buggle extends Buggle {
```

```
    public void box (int n) {
       forward(n-1);
       left();
       forward(n-1);
       left();
       forward(n-1);
       left();
       forward(n-1);
       left();
    }
}
```

The first thing you might notice about this version of `box` is that I left out `this`. When you invoke one method inside another, Java understands that you want to invoke it on the current object.

The next thing you should have noticed is the variable declaration `int n`, which is the parameter list. It says that this version of `box` expects a single argument with type `int`.

If we try to invoke `box` with a non-integer expression as an argument, or more than one argument, we get a compile-time error. As usual, you should try it out so you see what the error message looks like.

Now when we invoke `box` we have the option of including an argument:

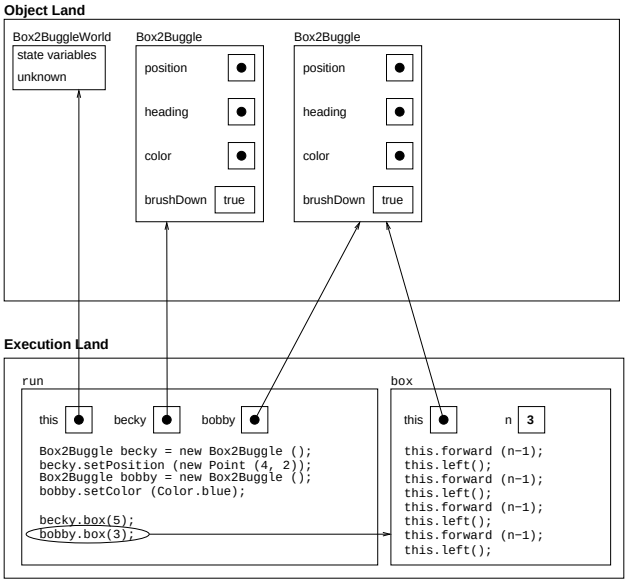
```
public class Box2World extends BuggleWorld {

    public void run () {
       Box2Buggle becky = new Box2Buggle ();
       becky.setPosition(new Point(4,2));
       Box2Buggle bobby = new Box2Buggle();
       bobby.setColor(Color.blue);

       becky.box(5);
       bobby.box(3);
    }
}
```

As an exercise, draw a picture of what BuggleWorld will look like after this code runs.

Here is what the JEM looks like while `bobby` is executing `box`:



We added the parameter `n` to the execution frame for `box`. The first time `box` executes, `this` refers to `becky` and `n` has the value 5. The second time, shown in the JEM, `this` refers to `bobby` and `n` is 3.

3.9 Local variables

In this example, the variable `n` only exists inside the method `box`. You cannot refer to it from `run`. Similarly, the variables `becky` and `bobby` only exist in `run` and you cannot refer to them from `box`.

Variables like these are called **local variables**. There are other variables, called **global variables**, that you can refer to from any method, but we will not see them for a while.

When a method completes, its execution frame disappears along with its local variables. Thus, each method is carefully isolated from the rest of the program. The only information that comes into the method is the parameter list; the only information that goes out is the return value. This isolation is one form of **modularity**. If methods are modular, it makes it easier to combine them and understand how they interact. As we write more complex programs, the benefits of modularity will become clearer.

3.10 Flow of execution

Flow of execution refers to the order statements are executed, and the order expressions are evaluated. In general, the body of a method is executed one statement at a time, starting at the top and proceeding until the end.

Within each statement, expressions are evaluated according to the rules of precedence. We have already talked about the order of evaluation for arithmetic expressions, and we have seen nested method invocations like

```
bobby.setColor (bobby.getColor().darker());
```

Expressions involving method invocations are evaluated from the innermost parentheses out, and from left to right.

When you invoke one of the built-in methods, you don't (and probably shouldn't) think about how the method does what it does. But when you are invoking one of your own methods, it is tempting to think about the flow of execution.

A method invocation is like a detour in the flow of execution. When the method is invoked, Java creates a new execution frame and starts executing the new method. At the same time it remembers where it is in the original frame. Again, the statements in the new method are executed one at a time, from top to bottom. When the method completes, the flow of execution returns to the original frame and picks up where it left off.

3.11 Picture objects

The Picture class defines a new type called `Picture` and a set of methods you can invoke on Picture objects. The Picture contract is in the Appendix.

Most of the methods that operate on Pictures are fruitful. For example, `clockwise90` takes a Picture as a parameter and returns a new Picture that is similar to the original, but rotated 90 degrees clockwise.

Assume that we have a Picture object named `original`. We can create a new object named `rotated`:

```
Picture rotated = clockwise90 (original);
```

Other methods flip pictures and juxtapose pictures in various ways, but it is worth empasizing that none of these methods modifies an existing picture in any way. They only create new Pictures.

3.12 Writing fruitful methods

Just as we extended Buggle and BuggleWorld to add new methods, we can extend Picture. The difference is that the Picture methods will be fruitful.

The following method takes four Pictures as parameters and returns a new Picture that contains all four pictures in a two-by-two grid.

```
public Picture fourPics (Picture p1, Picture p2, Picture p3, Picture p4){
    Picture row1 = beside (p1, p2);
    Picture row2 = beside (p3, p4);
    Picture grid = above (row1, row2);
```

```
    return grid;
}
```

The first line assembles the first two pictures into a row. The second line does the same with the last two pictures. The next line combines the rows into a grid.

The last line is a **return statement**. When `return` is executed, the method ends immediately and the flow of execution returns to the previous frame. For fruitful methods, the `return` statement has to include an expression. This expression gets evaluated and the result becomes the return value of the method.

Whoever invokes this method will get the object that `grid` refers to as a return value. We can make this method more concise by eliminating the temporary variables.

```
public Picture fourPics (Picture p1, Picture p2, Picture p3, Picture p4){
    return above (beside (p1, p2), beside (p3, p4));
}
```

The expression after the return statement is more complicated, but the effect is the same. Java evaluates the expression and returns the result.

When you declare a fruitful method, you have to declare the return type (in this case `Picture`). The expression in the return statement has to be the same type, otherwise Java reports a compile-time error. Try it out so you know what the error message looks like.

As an exercise, write a method called `fourSame` that takes a Picture as a parameter and that returns a new Picture with the original picture in all four quadrants of a grid.

3.13 Glossary

print statement: An invocation of `System.out.println` or `System.out.print` to display a value in the console window.

concatenate: To join two operands end-to-end.

console: A window that contains the text output of a program.

extend: Define a new class based on an existing class.

override: Write a new method for an extended class that replaces an existing method from the original class.

Java Execution Model (JEM): A visual language for representing the state of an executing Java program.

Object Land: The section of the JEM that represents objects.

Execution Land: The section of the JEM that represents methods as they execute and their local variables.

local variables: A variable that is declared inside a method and that exists only while the method is executing.

global variables: A variable that is declared outside all methods and that can be accessed from any method.

parameters: The variables that are used by a method to store the values that are passed in as arguments.

aliasing: The condition where more than one variable contains a reference to the same object.

modularity: The desirable property of a method that is well isolated from other methods.

return statement: A statement that terminates the current method and returns control to the previous frame. Inside a fruitful method the return statement determines the return value.

return type: The type of expression a method returns when it is invoked.

Chapter 4

Conditionals

4.1 Floating-point

There is only one problem with the integer arithmetic we have been using—no fractions. The solution is floating-point numbers, which can represent fractions as well as integers. In Java, the floating-point type is called `double` (short for double-precision floating-point).

You can create floating-point variables and assign values to them using the same syntax we used for the other types. For example:

```
double pi;
pi = 3.14159;
```

Of course, you can also declare the variable and assign a value at the same time:

```
int x = 1;
double pi = 3.14159;
```

Although `doubles` are useful, they are sometimes a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value `1`, is that an integer, a floating-point number, or both?

Strictly speaking, Java distinguishes the integer value `1` from the floating-point value `1.0`, even though they seem to be the same number. They belong to different types, and you are not allowed to make assignments between types. For example, the following is illegal:

```
int x = 1.1;
```

because the variable on the left is an `int` and the value on the right is a `double`. But it is easy to forget this rule, especially because there are places where Java will automatically convert from one type to another. For example,

```
double y = 1;
```

should technically not be legal, but Java allows it by converting the `int` to a `double` automatically. This leniency is convenient, but it can cause problems; for example:

```
double y = 1 / 3;
```

You might expect the variable `y` to be given the value `0.333333`, which is a legal floating-point value, but in fact it will get the value `0.0`. The reason is that the expression on the right is the ratio of two integers, so Java does *integer* division, which yields the integer value `0`. Converted to floating-point, the result is `0.0`.

One way to solve this problem (once you figure out what it is) is to make the right-hand side a floating-point expression:

```
double y = 1.0 / 3.0;
```

This sets `y` to `0.333333`, as expected.

All the operations we have seen so far—addition, subtraction, multiplication, and division—also work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

4.2 Converting from `double` to `int`

As I mentioned, Java converts `int`s to `double`s automatically if necessary, because no information is lost in the translation. On the other hand, going from a `double` to an `int` requires rounding off. Java doesn't perform this operation automatically, in order to make sure that you, as the programmer, are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **typecast**. Typecasting is so called because it allows you to take a value that belongs to one type and "cast" it into another type (in the sense of molding or reforming, not throwing).

The syntax for typecasting is ugly: you put the name of the type in parentheses and use it as an operator. For example,

```
int x = (int) 3.14159;
```

The `(int)` operator has the effect of converting what follows into an integer, so `x` gets the value 3.

Typecasting takes precedence over arithmetic operations, so in the following example, the value of `pi` gets converted to an integer first, and the result is 60, not 62.

```
double pi = 3.14159;
int x = (int) Math.PI * 20.0;
```

Converting to an integer always rounds down, even if the fraction part is 0.99999999.

These two properties (precedence and rounding) can make typecasting awkward.

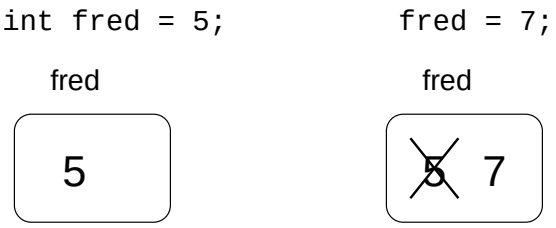
4.3 Multiple assignment

I haven't said much about it, but it is legal in Java to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value.

```
int fred = 5;
System.out.print (fred);
fred = 7;
System.out.println (fred);
```

The output of this program is `57`, because the first time we print `fred` his value is 5, and the second time his value is 7.

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:



When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality.

In mathematics, a statement of equality is true for all time. If $a = b$ now, then a will always equal b. In Java, an assignment statement can make two variables equal, but they don't have to stay that way!

```
int a = 5;
int b = a;     // a and b are now equal
a = 3;         // a and b are no longer equal
```

The third line changes the value of `a` but it does not change the value of `b`, and so they are no longer equal.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.

When a program contains multiple assignments to the same variable, it is tempting, out of habit, to declare the type of the variable every time. For example,

```
int fred = 3;

// and then later ...

int fred = 5;
```

This code is illegal because it tries to declare (and create) the variable `fred` twice. After the initial declaration, all subsequent assignments omit the variable type.

4.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if (x > 0) {
    System.out.println ("x is positive");
}
```

The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the **comparison operators**:

```
x == y              // x equals y
x != y              // x is not equal to y
x > y               // x is greater than y
x < y               // x is less than y
x >= y              // x is greater than or equal to y
x <= y              // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax Java uses is a little different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single `=` instead of a double `==`. Remember that `=` is the assignment operator, and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

The two sides of a condition have to be the same type, so we can only compare `int` to `int` and `double` to `double`. You cannot use the comparison operators on objects, so there is no way to know which of two Buggles is greater.

4.5 Alternative execution

A second form of conditional execution is **alternative execution** in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if (x > 0) {
    System.out.println ("x is positive");
} else {
    System.out.println ("x is negative or zero");
}
```

If the condition is true, that means that `x` is positive, and this code prints a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

4.6 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if`s and `else`s:

```
if (x > 0) {
    System.out.println ("x is positive");
} else if (x < 0) {
    System.out.println ("x is negative");
} else {
    System.out.println ("x is zero");
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-brackets lined up, you are less likely to make syntax errors and you will find them more quickly if you do.

4.7 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if (x == 0) {
    System.out.println ("x is zero");
} else {
    if (x > 0) {
        System.out.println ("x is positive");
    } else {
        System.out.println ("x is negative");
    }
}
```

There is now an outer conditional that contains two branches. The first branch contains a simple `print` statement, but the second branch contains another conditional statement, which has two branches of its own. Fortunately, those two branches are both `print` statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

4.8 Boolean expressions

Most of the operations we have seen produce results that are the same type as their operands. For example, the `+` operator takes two `int`s and produces an `int`, or two `double`s and produces a `double`, etc.

The exceptions we have seen are the comparison operators, which compare values and return either `true` or `false`. `true` and `false` are special values in Java, and together they make up a type called **boolean**.

Boolean expressions and variables work just like other types of expressions and variables:

```
boolean fred;
fred = true;
boolean testResult = false;
```

The first example is a simple variable declaration; the second example is an assignment, and the third example is a combination of a declaration and an assignment. The values `true` and `false` are keywords in Java, so they may appear in a different color, depending on your development environment.

As I mentioned, the result of a conditional operator is a boolean, so you can store the result of a comparison in a variable:

```
boolean positiveFlag = (x > 0);    // true if x is positive
```

and then use it as part of a conditional statement later:

```
if (positiveFlag) {
    System.out.println ("x was positive when I checked it");
}
```

A variable used in this way is called a **flag**, since it flags the presence or absence of some condition.

4.9 Logical operators

There are three **logical operators** in Java: AND, OR and NOT, which are denoted by the symbols `&&`, `||` and `!`. The semantics (meaning) of these operators is similar to their meaning in English. For example `x > 0 && x < 10` is true only if `x` is greater than zero AND less than 10.

`positiveFlag || x<0 == 0` is true if *either* of the conditions is true, that is, if `positiveFlag` is true OR `x` is less than zero.

Finally, the NOT operator has the effect of negating or inverting a boolean expression, so `!positiveFlag` is true if `positiveFlag` is false.

Logical operators often provide a way to simplify nested conditional statements. For example, how would you write the following code using a single conditional?

```
if (x > 0) {
  if (x < 10) {
    System.out.println ("x is a positive single digit.");
  }
}
```

4.10 Boolean methods

Methods can return boolean values just like any other type. The Buggle class includes several boolean methods:

`public boolean isBrushDown ()`

Return true if this Buggle will leave a trail when it moves, false otherwise.

`public boolean isOverBagel ()`

Returns true if there is a bagel in the cell currently occupied by this buggle, false otherwise.

`public boolean isFacingWall ()`

Returns true if this buggle is next to a wall of Buggle world and facing it, false otherwise.

It is common to give boolean methods names that sound like yes/no questions. These methods are useful for checking whether an operation is possible before trying it. For example:

```
if (becky.isOverBagel ()) {
    becky.pickUpBagel ();
}
```

You can write your own boolean methods. For example, Buggles do not provide a method named `isBrushUp`, but you could write your own.

```
public boolean isBrushUp () {
    if (isBrushDown() == true) {
        return false;
    } else {
        return true;
    }
}
```

The return type is `boolean`, which means that every return statement has to provide a boolean expression.

This code is nice and readable, although it is a bit longer than it needs to be. The return value from the original method, `isBrushDown`, has type boolean, so all we have to do is apply the NOT operator.

```
public boolean isBrushUp () {
    return !isBrushDown ();
}
```

Here is a more interesting boolean method. It checks whether there is a bagel on the square one step in front of the Buggle.

```
public boolean isBagelInFront () {
    if (isFacingWall ()) {
        return false;
    } else {
        forward ();
        boolean result = isOverBagel ();
        backward ();
        return result;
    }
}
```

There are two (slightly) tricky things here. One is the first branch of the conditional. If we are facing a wall, then we know there is no bagel there. In that case we can return `false` immediately. This kind of check is sometimes called a **special case**.

In the common case, all we have to do is move forward (which we now know is safe) and check for a bagel. The second tricky thing is that we have to store the result temporarily because we have to move the Buggle back before we get to the `return` statement.

The following code changes the state of the brush from up to down or down to up:

```
    if (becky.isBrushDown ()) {
        becky.brushUp ();
    } else {
        becky.brushDown ();
    }
```

Flipping a boolean value like this is called **toggling**.

4.11 Variables inside conditionals

A variable declared inside an if statement only exists inside the if statement.

```
public boolean isBagelInFront () {
    if (isFacingWall ()) {
        boolean result = false;
    } else {
        forward ();
        boolean result = isOverBagel ();
        backward ();
    }
    return result;    // WRONG!!!
}
```

In this example, both branches create variables named `result`, but outside the conditional, neither one exists.

The easiest way to fix this is to put return statements into both branches of the conditional (as in the previous version of this method).

The alternative is to declare the variable outside the conditional.

```
public boolean isBagelInFront () {
    boolean result;
    if (isFacingWall ()) {
        result = false;
    } else {
        forward ();
        result = isOverBagel ();
        backward ();
    }
    return result;    // this works
}
```

We can simplify this version a little by initializing the result to false and then changing it only if conditions warrant it.

```
public boolean isBagelInFront () {
    boolean result = false;
    if (!isFacingWall ()) {
        forward ();
        result = isOverBagel ();
        backward ();
    }
    return result;    // this works
}
```

As always when there are multiple ways to express a computation, you should choose the one that is easiest to read and that reflects the natural structure of the problem.

4.12 Glossary

typecast: Convert a value from one type to another.

conditional statement: Any of the forms of an `if` statement that execute statements (or not) depending on the result of a boolean expression.

alternative execution: A conditional statement with two branches, an if branch and an else branch, exactly one of which will be executed.

chaining: A way of organizing a string of consecutive conditional statements.

nested structure: In general, the ability to nest one statement within another. In the context of conditional statements, nesting is one way of organizing a set of conditional statements.

comparison operators: Any of the operators that compare values and produce boolean values.

boolean: A primitive type in Java, made up of the values `true` and `false`.

flag: A boolean variable that record the result of a condition operator to "flag" a condition.

logical operators: The operators AND, OR, and NOT, which are used to combine boolean expressions.

special case: A part of a program that handles unusual conditions that cannot be handled by the general case.

general case: A part of a program that handles the most common and extensive set of conditions.

toggle: To flip the state of a boolean variable from false to true and vice versa.

Chapter 5

Recursion

Now that we have frutiful methods and condition statements, you might be interested to know that we have a **complete** programming language, by which I mean that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features we have used so far (actually, we would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a non-trivial exercise first accomplished by Alan Turing, one of the first computer scientists (well, some would argue that he was a mathematician, but a lot of the early computer scientists started as mathematicians). Accordingly, it is known as the Turing thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

5.1 Recursion

I mentioned in the last chapter that it is legal for one method to invoke another, and we have seen several examples of that. I neglected to mention that it is also legal for a method to invoke itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following method:

```
public void countdown (int n) {
    if (n == 0) {
        System.out.println ("Blastoff!");
    } else {
        System.out.println (n);
        countdown (n-1);
    }
}
```

The name of the method is `countdown` and it takes a single integer as a parameter. If the parameter is zero, it prints the word "Blastoff." Otherwise, it prints the number and then invokes a method named `countdown`—itself—passing `n-1` as an argument.

What happens if we invoke the method like this:

```
countdown (3);
```

The execution of `countdown` begins with `n=3`, and since `n` is not zero, it prints the value 3, and then invokes itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not zero, it prints the value 2, and then invokes itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not zero, it prints the value 1, and then invokes itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is zero, it prints the word "Blastoff!" and then returns.
> >
> > The countdown that got `n=1` returns.
>
> The countdown that got `n=2` returns.

The countdown that got `n=3` returns.

And then you're back to the original invocation of `countdown`. So the total output looks like:

```
3
2
1
Blastoff!
```

The process of a method invoking itself is called **recursion**, and such methods are said to be **recursive**.

5.2 Recursive definition

A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is typically not very useful:

frabjuous: an adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function **factorial**, you might get something like:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

(Factorial is usually denoted with the mathematical symbol !, which is not to be confused with the Java logical operator `!` which means NOT.) This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$. So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, we get 3! equal to 3 times 2 times 1 times 1, which is 6.

5.3 Fruitful recursive methods

If you can write a recursive definition of something, you can usually write a Java program to evaluate it. The first step is to decide what the parameters are for the function, and what the return type is. With a little thought, you should conclude that factorial takes an integer as a parameter and returns an integer:

```
public int factorial (int n) {
}
```

You can add this method to the Buggle class to try it out. Buggles enjoy doing arithmetic in their spare time.

If the argument happens to be zero, all we have to do is return 1:

```
public int factorial (int n) {
    if (n == 0) {
        return 1;
    }
}
```

Otherwise, and this is the interesting part, we have to make a recursive invocation to find the factorial of $n-1$, and then multiply it by n.

```
public int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        int recurse = factorial (n-1);
        int result = n * recurse;
        return result;
    }
}
```

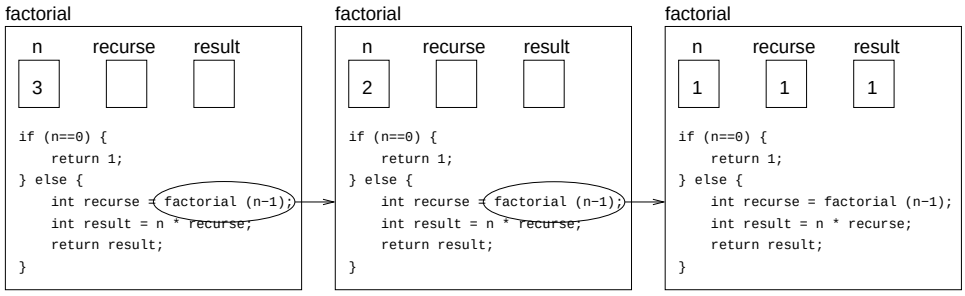
If we invoke `factorial` with the value 3:

Since 3 is not zero, we take the second branch and calculate the factorial of $n-1$...

> Since 2 is not zero, we take the second branch and calculate the factorial of $n - 1$...
>
> > Since 1 is not zero, we take the second branch and calculate the factorial of $n - 1$...
> >
> > > Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive invocations.
> >
> > The return value (1) gets multiplied by `n`, which is 1, and the result is returned.
>
> The return value (1) gets multiplied by `n`, which is 2, and the result is returned.

The return value (2) gets multiplied by `n`, which is 3, and the result, 6, is returned to whoever invoked `factorial (3)`.

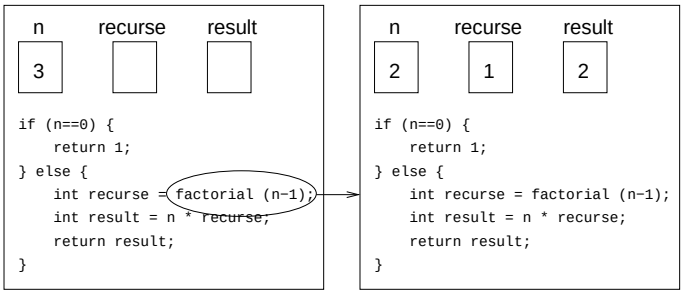
Here is what Execution Land looks like while this recursion is taking place.



This is a snapshot of the point in the execution where the last invocation of `factorial`, the one that got 0 as an argument, has just returned the value 1.

The previous invocation of `factorial` takes that result and multiplies it by `n`, which is 1, yielding the `result` 1. That's the value it is about to return.

If we come back a little later, we will see this state:



The execution frame that got `n=1` has completed and vanished. But before it left, it returned the value 1, which was assigned to the variable `recurse`. The execution frame that got `n=2` can now compute `result=2` and return 2.

As an exercise, draw a JEM for the program state just before the last execution frame disappears.

5.4 Leap of faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, it can quickly become labarynthine. An alternative is what I call the "leap of faith." When you come to a method invocation, instead of following the flow of execution, you *assume* that the method works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in methods. When you invoke a built-in method, you don't examine its implementation. You just assume that it works, because the people who wrote the Java classes were good programmers.

Well, the same is true when you invoke one of your own methods. For example, as an exercise, write a method called `isSingleDigit` that determines whether a number is between 0 and 9. Once you have convinced ourselves that this method is correct—by testing and examination of the code—you can use the method without ever looking at the code again.

The same is true of recursive methods. When you get to the recursive invocation, instead of following the flow of execution, you should *assume* that the recursive invocation works (yields the correct result), and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it is a bit strange to assume that the method works correctly when you have not even finished writing it, but that's why it's called a leap of faith!

5.5 Another example

In the previous example I used temporary variables to spell out the steps, and to make the code easier to debug, but I could have saved a few lines:

```
public int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial (n-1);
    }
}
```

From now on I will tend to use the more concise version, but I recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up, if you are feeling inspired.

After `factorial`, the classic example of a recursively-defined mathematical function is `fibonacci`, which has the following definition:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);$$

Translated into Java, this is

```
public int fibonacci (int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fibonacci (n-1) + fibonacci (n-2);
    }
}
```

If you try to follow the flow of execution here, even for fairly small values of `n`, your head explodes. But according to the leap of faith, if we assume that the two recursive invocations (yes, you can make two recursive invocations) work correctly, then it is clear that we get the right result by adding them together.

5.6 Recursion in BuggleWorld

Let's apply what we've learned about recursion to a Buggle problem. What if we would like to tell a Buggle to go forward until it hits a wall? As long as Buggles know when they reach the wall, we can do this with recursion. Fortunately, Buggles provide a method called `isFacingWall` that returns `true` if the Buggle is next to a wall and facing it.

Assume that the Buggle is three steps from the wall, as in the figure:

The solution starts by checking if we are already facing a wall. If so, then the problem is trivial—we can return immediately without doing anything. Otherwise, we should take a single step forward. Here's what the program looks like so far:

```
public void goToWall () {
    if (isFacingWall ()) {
        return;
    } else {
        forward ();
```

```
        // now what?
    }
}
```

And then what? Well, we are right back where we started. We want the Buggle to go forward until it reaches a wall. Fortunately, we know how to do that. All we have to do is invoke `goToWall`:

```
public void goToWall () {
    if (isFacingWall ()) {
        return;
    } else {
        forward ();
        goToWall ();
    }
}
```

Strangely, that's all there is to it. As an exercise, draw a JEM for this method when the last execution frame is about to complete.

5.7 Recursion with parameters

By now you have seen TurtleWorld and Turtles in lecture and laboratory. We can extend the Turtle class and add the following method:

```
public void spiral (int steps, int angle, int length, int increment) {
    if (steps==0) {
        return;
    } else {
        fd (length);
        lt (angle);
        spiral (steps-1, angle, length+increment, increment);
    }
}
```

`spiral` takes four integer parameters. The first parameter, `steps` determines how many line segments there are in the spiral. When `steps` is zero, we are done—the method does nothing and returns without making a recursive invocation. This is called the **base case**, because it is the end of the recursion. The other branch of the conditional is called the **recursive step**, since it takes one step toward the base case.

In this case the recursive step draws a line segment with length `length`, turns to the left by angle `angle` and then makes a recursive invocation to draw the rest of the spiral.

The arguments of the recursive invocation are based on the parameters we just received. Since we just took one step, the number of steps remaining is `steps-1`. The angle between segments doesn't change, so we just pass it along.

The value we pass as the new `length` is the old length plus whatever the value of `increment` is. Changing the length from step to step is what makes a spiral. Otherwise, we get a ... well, you can figure it out, or try it by setting `increment` to zero.

It might seem wasteful to pass `angle` and `increment` from one execution frame to the next, since they don't change. But that's the price we pay for modularity.

5.8 Recursion with return values

Suppose we want to create a Buggle that walks to the wall eating bagels as she goes along and returns the number of bagels that she has eaten. How can we do this? Well, if we only had to walk to the wall eating bagels this would be easy. All we would have to do is make the Buggle go to the wall (as we just learned in the previous section) and along the way, if she is over any bagels, have her eat them. So, the new piece of this problem is to have her count the number of bagels as she eats them and return the result.

Here is a method that solves this problem:

```
public int eatBagels () {
    if (isFacingWall ()) {
        if (isOverBagel ()) {
            pickUpBagel ();
            return 1;
        } else {
            return 0;
        }
    }

    if (isOverBagel ()) {
        pickUpBagel ();
        forward ();
        return eatBagels () + 1;
    } else {
        forward ();
        return eatBagels ();
    }
  }
}
```

The return type of the method is `int`. It returns the number of bagels that got eaten. As an exercise, draw the JEM for this example.

5.9 Infinite recursion

In all the examples we have looked at, there is a branch in each recursive method that returns without making another recursive call. This branch is the base case, and it is required to make the program finish in a finite amount of time. Without it, the program will recurse forever.

```
public void countdown (int n) {
   System.out.println (n);
   countdown (n-1);
}
```

This version of countdown is recursive, but it has no base case. If you run it, you see something like:

```
3
2
1
0
-1
-2
-3

...
```

And eventually you will see a `StackOverflowException`. The **stack** is the structure in the Java run-time system that contains all the execution frames for running methods. In a recursive method, there is one execution frame for every time the method invokes itself.

If a method recurses forever, eventually the stack fills with execution frames and overflows. At that point the program stops running. This situation is called an **infinite recursion**.

In general, to show that a recursive method is correct, you have to show three things:

- There is a base case (and it is correct).
- The recursive step is correct, assuming that the recursive invocation works.
- For all parameters the method might receive, the program will reach the base case in a finite number of steps.

5.10 Glossary

complete language: A programming language that can implement any computable function.

recursion: A way of programming that involves methods that invoke themselves.

recursive: A program that contains one or more recursive methods.

base case: The branch in a recursive function that does not make a recursive invocation, thereby terminating the recursion.

recursive step: The branch in a recursive function that makes one or more recursive invocations.

stack: The structure in the Java run-time system that contains execution frames.

infinite recursion: If a recursion lacks a base case, or the base case is never reached, the program will continue to recurse forever, or until a run-time error occurs.

Chapter 6

Lists

6.1 The modulus operator

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In Java, the modulus operator is a percent sign, `%`. The syntax is exactly the same as for other operators:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y`.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, `x % 10` yields the rightmost digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

6.2 Integer Linked Lists

A **linked list** is a compound data structure, which means that it contains an ordered set of elements. The IntList class defines a particular kind of linked list whose elements are integers.

Lists are **recursive data structures**, because they are defined recursively, like this.

An IntList is either:

- the empty list, or
- a node made up of a head and a tail. The head is an integer and the tail is an IntList.

This definition is recursive in the same way the definition of "frabjuous" is.

As usual, the IntList contract describes the methods you use to create and manipulate lists.

6.3 IntList methods

There are no IntList constructors, and no methods you can invoke on objects. In other words, there are no **instance methods**. Instead, all IntList methods are **class methods**.

The first methods we'll look at are `empty` and `prepend`.

> `public static IntList empty()`
>
> Returns an empty integer list.
>
> `public static IntList prepend (int n, IntList list)`
>
> Returns a new integer list node whose head is `n` and whose tail is `list`.

The word `static` in the function declaration indicates that it is a class method. If you refer to an IntList method from inside another class definition, you have to specify the IntList class:

```
IntList elist = IntList.empty ();
```

This statement creates a new variable and makes it point to a new object that represents an empty list.

The `prepend` method takes an element and an IntList as arguments and returns a new list that has all the elements of the original list, plus the new element prepended (added at the beginning).

```
IntList singleton = IntList.prepend (4, elist);
```

The result is that `singleton` refers to a list that contains the element 4. A list with a single element is called a **singleton**.

There is a standard way to represent lists using box and pointer notation:

The variable named `list` is represented in the usual way; it contains a reference to the first node in the list. The nodes are represented as boxes containing an element (the head) and a reference to the next node (the tail). At the end of the list, the empty list is represented by a box with a dot and no element.

6.4 Math methods

The Math class is a built-in class (part of the Java language) that provides all the basic math functions, like `sin` and `cos`. They are all class methods, so when you invoke them you have to specify the name of the class and the name of the method:

```
double root = Math.sqrt (17.0);
double angle = 1.5;
double height = Math.sin (angle);
```

The first example sets `root` to the square root of 17. The second example finds the sine of `1.5`, which is the value of the variable `angle`. Java assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π. Conveniently, Java provides π as a built-in value:

```
double degrees = 90;
double angle = degrees * 2 * Math.PI / 360.0;
```

Notice that `PI` is in all capital letters. Java does not recognize `Pi`, `pi`, or `pie`.

Another useful method in the `Math` class is `round`, which rounds a floating-point value off to the nearest integer and returns an `int`.

```
int x = Math.round (Math.PI * 20.0);
```

In this case the multiplication happens first, before the method is invoked. The result is 63 (rounded up from 62.8319).

Finally, the Math class contains `max` and `min` functions that return the larger of two numbers. It works with both integers and doubles.

```
int bigger = Math.max (5, 7);
double smaller = Math.min (5.5, 7.7);
```

What do you think happens if you invoke `Math.max (5, 7.7)`?

6.5 Extending IntList

If you extend the IntList class, you can refer to the IntList methods by name, without invoking them on an object. For example:

```
public class MyList extends IntList {

    public static MyList singleton (int n) {
        IntList elist = empty ();
        IntList singleton = prepend (n, elist);
        return singleton;
    }
}
```

The MyList class has all the same methods as IntList, but in addition there is a method called `singleton` that takes an integer and returns a MyList object that represents a list with a single element.

Since MyList extends IntList, Java knows that `empty` refers to the `IntList.empty` method.

6.6 Anatomy of an IntList

Every (non-empty) list is made up of a head and a tail. The head is the element of the first node. The tail is a list that contains all but the first node. The IntList class contains methods to extract the head and tail of lists:

```
int first = head (list);
```

If you invoke `head` and pass an empty list as an argument, it causes a run-time error. You can avoid that by checking for an empty list before invoking head:

```
if (!isEmpty (list)) {
    int first = head (list);
}
```

The `isEmpty` method returns `true` if the list is empty, and `false` otherwise.

To get the tail of a list:

```
IntList rest = tail (list);
```

Again, it is illegal to send an empty list as an argument to `tail`.

It is tempting to think of `head` and `tail` as symmetric operations, but remember that their return types are not the same.

6.7 Printing IntLists

While you are debugging, it is useful to be able to print the elements of a list. It is not obvious how to do that, but we can start by thinking about what we do know. First, we know how to print the empty list—do nothing. Also, we know how to print the head of the list, since we know how to print integers:

```
    int first = head (list);
    System.out.println (first);
```

All we need now is a way to print the rest of the list. Using the divide, conquer and glue strategy, we can solve the problem recursively:

```
public static void printList (IntList list)
{
    if (isEmpty (list)) {
        return;
```

```
    }
    int first = head (list);
    System.out.println (first);
    IntList rest = tail (list);
    printList (rest);
}
```

The base case is the empty list; it returns without doing anything. For the remainder of the method, we know that the list cannot be empty, because otherwise we would have returned already. Therefore we know that the invocations of `head` and `tail` cannot cause an error.

The rest of the method is straightforward. We use `println` to print the head of the list and `printList` to print the tail.

Processing a list like this, by performing an operation on the head and then processing the tail, is called **traversing** or **walking down** the list.

6.8 Traversing lists

Traversing a list is a general tool that solves a lot of problems. For example, the following method adds up all the elements in a list:

```
public static int sum (IntList list)
{
    if (isEmpty (list)) {
        return 0;
    }
    int first = head (list);
    IntList rest = tail (list);
    int sumRest = sum (rest);
    return first + sumRest;
}
```

Again, we solve the problem by identifying a base case and a recursive step. The base case is an empty list; we know that the sum of no elements is 0, so we can return immediately without making a recursive invocation.

The recursive step has three parts: first, we break the list into a head and a tail. Next, we find the sum of the elements in the head (that's trivial). Finally, we find the sum of the elements in the tail (that's the recursive invocation).

The grand total is just the sum of `first` and `sumRest`. We can write this method more concisely by eliminating the temporary variables:

```
public static int sum (IntList list)
{
    if (isEmpty (list)) {
        return 0;
    } else {
```

```
        return head (list) + sum (tail (list));
    }
}
```

As an exercise, write a method called `prod` that computes the product of all the elements in a list (multiplies them all together).

The reason this solution works is that addition is associative. That implies that the sum of all the elements is the same as the first element plus the sum of the rest.

6.9 Checking lists

The same logic applies to traversing a list and checking whether a condition holds. For example, `areAllPositive` traverses a list and checks if all the elements are positive.

As usual, the base case is the empty list, although it takes a bit of thought to decide whether a list with no elements can be considered all positive. It turns out that it's best to say "yes." After all, it certainly doesn't contain any negative elements.

In addition to the case of the empty list, this method has a second base case. If we find a non-positive element, we can stop immediately and return false without making a recursive invocation and without traversing the rest of the list.

Finally, the recursive step goes like this: if the current element is positive, and the rest of the list is all positive, then the whole list must be positive.

```
public static boolean areAllPositive (IntList list)
{
    if (isEmpty (list)) {
        return true;
    } else {
        if (head (list) <= 0) {
            return false;
        } else {
            return areAllPositive (tail (list));
        }
    }
}
```

I am not fond of nested conditionals, so I would consider rewriting the method like this:

```
public static boolean areAllPositive (IntList list)
{
    if (isEmpty (list)) {
        return true;
```

```
    }
    return (head(list) > 0) && areAllPositive (tail (list));
}
```

6.10 Filtering a list

As a final example, we will traverse a list and create a new list that contains only the even elements from the original. This kind of selection is called **filtering**.

As usual, we approach the problem by identifying a base case and a recursive step. The base case is—you guessed it—the empty list. Obviously the empty list contains no even elements, so the result is the empty list.

As usual, the next step is to break the list into a head and a tail. The recursive step is to find all the even elements in the tail.

Now there are two possibilities. If the element in the head is odd, then all we have to do is return the list of even numbers we got from the recursive invocation.

If the head element is even, then we have to prepend it onto the list of even number. Fortunately, we have a method for that.

Here's what it all looks like in Java.

```
public static IntList allEven (IntList list)
{
    if (isEmpty (list)) {
        return empty ();
    }
    IntList evens = allEven (tail (list));

    int first = head (list);
    if (first%2 == 0) {
        return prepend (first, evens);
    } else {
        return evens;
    }
}
```

There are three `return` statements, but that's ok, because only one of them can ever execute, and all of them return an expression that has the right type.

As an exercise, write a method called `postpend` that takes an integer and a list, and that returns a new list with all the elements of the original plus the integer added at the end.

6.11 ObjectLists

The ObjectList class is similar to the IntList class except that the elements can be any kind of object. For example, to build a list of Buggles:

```
ObjectList list = ObjectList.empty ();

list = ObjectList.prepend (becky, list);
list = ObjectList.prepend (bobby, list);
```

In this example I make multiple assignments to the variable `list`. Each assignment changes the reference to refer to a new list. In the end, `list` refers to a list that contains two Buggles as elements.

The elements in the list have type `Object`, so the return type from `head` is also `Object`:

```
Object obj = ObjectList.head (list);
```

The object that `obj` refers to is `bobby`, but at this point the compiler does not know that `bobby` is a Buggle, so we can't use any of the Buggle methods. Try the following so you will see what the error message looks like:

```
    obj.forward ();      // this is not legal
```

It's as if `bobby` has amnesia. Fortunately, we can remind him of his Buggleness with a typecast:

```
Buggle bobby = (Buggle) obj;
bobby.forward();
```

If we made a mistake somehow, and the object that `obj` refers to is not actually a Buggle, we get a `ClassCastException`.

Other than that bit of awkwardness, ObjectLists work just the way IntLists do.

6.12 Glossary

linked list: A compound data structure made up of a sequence of nodes, each of which contains an element and a reference to the next node.

instance method: A method that is invoked on an object.

class method: A method that is not invoked on an object.

singleton: A list with a single element.

traverse: An operation that processes a list by applying an operating to each element in the list.

walk down: Another way of saying "traverse".

filter: A way of processing a compound data structure and selecting elements that have a certain property.

Chapter 7

Iteration

7.1 The `while` statement

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have already seen programs that use recursion to perform repetition, like `countdown` and `spiral`. This type of repetition is called **iteration**, and Java provides several language features, like the `while` statement, that make it easier to write iterative programs.

Using a `while` statement, we can rewrite `countdown`:

```
public void countdown (int n) {
    while (n > 0) {
        System.out.println (n);
        n = n-1;
    }
    System.out.println ("Blastoff!");
}
```

You can almost read a `while` statement as if it were English. What this method means is, "While `n` is greater than zero, continue printing the value of `n` and then reducing the value of `n` by 1. When you get to zero, print the word 'Blastoff!"'

More formally, the flow of execution for a `while` statement is as follows:

1. Evaluate the condition in parentheses, yielding `true` or `false`.

2. If the condition is false, exit the `while` statement and continue execution at the next statement.

3. If the condition is true, execute each of the statements between the squiggly-brackets, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The statements inside the loop are sometimes called the **body** of the loop.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `countdown`, we can prove that the loop will terminate because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop (each **iteration**), so eventually we have to get to zero. In other cases it is not so easy to tell:

```
public void sequence (int n) {
    while (n != 1) {
        System.out.println (n);
        if (n%2 == 0) {           // n is even
            n = n / 2;
        } else {                  // n is odd
            n = n*3 + 1;
        }
    }
}
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which will make the condition false.

At each iteration, the program prints the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program will terminate. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even every time through the loop, until we get to 1.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it!

7.2 Tables

One of the things loops are good for is generating and printing tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand.

To make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the result tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly), but shortsighted. Soon thereafter computers (and calculators) were so pervasive that the tables became obsolete.

Well, almost. It turns out that for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division.

Although a "log table" is not as useful as it once was, it still makes a good example of iteration. The following program prints a sequence of values in the left column and their logarithms in the right column:

```
double x = 1.0;
while (x < 10.0) {
    System.out.println (x + "   " + Math.log(x));
    x = x + 1.0;
}
```

The output of this program is

```
1.0   0.0
2.0   0.6931471805599453
3.0   1.0986122886681098
4.0   1.3862943611198906
5.0   1.6094379124341003
6.0   1.791759469228055
7.0   1.9459101490553132
8.0   2.0794415416798357
9.0   2.1972245773362196
```

Looking at these values, can you tell what base the `log` function uses by default?

Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To find that, we have to use the following formula:

$$\log_2 x = \frac{log_e x}{log_e 2} \tag{7.1}$$

Changing the `print` statement to

```
System.out.println (x + "   " + Math.log(x) / Math.log(2.0));
```

yields

```
1.0   0.0
2.0   1.0
3.0   1.5849625007211563
4.0   2.0
5.0   2.321928094887362
6.0   2.584962500721156
7.0   2.807354922057604
8.0   3.0
9.0   3.1699250014423126
```

We can see that 1, 2, 4 and 8 are powers of two, because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
double x = 1.0;
while (x < 100.0) {
    System.out.println (x + "   " + Math.log(x) / Math.log(2.0));
    x = x * 2.0;
}
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a **geometric** sequence. The result is:

```
1.0   0.0
2.0   1.0
4.0   2.0
8.0   3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

Log tables may not be useful any more, but for computer scientists, knowing the powers of two is! Some time when you have an idle moment, you should memorize the powers of two up to 65536 (that's 2^{16}).

7.3 Two-dimensional tables

A two-dimensional table is a table where you choose a row and a column and read the value at the intersection. A multiplication table is a good example. Let's say you wanted to print a multiplication table for the values from 1 to 6.

A good way to start is to write a simple loop that prints the multiples of 2, all on one line.

```
int i = 1;
while (i <= 6) {
```

```
        System.out.print (2*i + "   ");
        i = i + 1;
    }
    System.out.println ("");
```

The first line initializes a variable named `i`, which is going to act as a counter, or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6, and then when `i` is 7, the loop terminates. Each time through the loop, we print the value `2*i` followed by three spaces. Since we are using the `print` command rather than `println`, all the output appears on a single line.

In some environments the output from `print` gets stored without being displayed until `println` is invoked. If the program terminates, and you forget to invoke `println`, you may never see the stored output.

The output of this program is:

```
2   4   6   8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

7.4 Encapsulation and generalization

Encapsulation usually means taking a piece of code and wrapping it up in a method, allowing you to take advantage of all the things methods are good for.

Generalization means taking something specific, like printing multiples of 2, and making it more general, like printing the multiples of any integer.

Here's a method that encapsulates the loop from the previous section and generalizes it to print multiples of `n`.

```
public void printMultiples (int n) {
  int i = 1;
  while (i <= 6) {
    System.out.print (n*i + "   ");
    i = i + 1;
  }
  System.out.println ("");
}
```

To encapsulate, all I had to do was add the first line, which declares the name, parameter, and return type. To generalize, all I had to do was replace the value 2 with the parameter `n`.

If I invoke this method with the argument 2, I get the same output as before. With argument 3, the output is:

```
3   6   9   12   15   18
```

and with argument 4, the output is

```
4   8   12   16   20   24
```

By now you can probably guess how we are going to print a multiplication table: we'll invoke `printMultiples` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

```
int i = 1;
while (i <= 6) {
  printMultiples (i);
  i = i + 1;
}
```

First of all, notice how similar this loop is to the one inside `printMultiples`. All I did was replace the print statement with a method invocation.

The output of this program is

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

which is a (slightly sloppy) multiplication table. If the sloppiness bothers you, Java provides methods that give you more control over the format of the output, but I'm not going to get into that here.

7.5 Methods

In the last section I mentioned "all the things methods are good for." About this time, you might be wondering what exactly those things are. Here are some of the reasons methods are useful:

- By giving a name to a sequence of statements, you make your program easier to read and debug.
- Dividing a long program into methods allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Methods facilitate both recursion and iteration.
- Well-designed methods are often useful for many programs. Once you write and debug one, you can reuse it.

7.6 More encapsulation

To demonstrate encapsulation again, I'll take the code from the previous section and wrap it up in a method:

```
public void printMultTable () {
  int i = 1;
  while (i <= 6) {
    printMultiples (i);
    i = i + 1;
  }
}
```

The process I am demonstrating is a common development plan. You develop code gradually by adding lines to an existing method, and then when you get it working, you extract it and wrap it up in a new method.

The reason this is useful is that you sometimes don't know when you start writing exactly how to divide the program into methods. This approach lets you design as you go along.

7.7 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to `printMultTable`:

```
public void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i);
    i = i + 1;
  }
}
```

I replaced the value 6 with the parameter `high`. If I invoke `printMultTable` with the argument 7, I get

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
7   14   21   28   35   42
```

which is fine, except that I probably want the table to be square (same number of rows and columns), which means I have to add another parameter to `printMultiples`, to specify how many columns the table should have.

Just to be annoying, I will also call this parameter `high`, demonstrating that different methods can have parameters with the same name (just like local variables):

```
public void printMultiples (int n, int high) {
  int i = 1;
  while (i <= high) {
    System.out.print (n*i + "   ");
    i = i + 1;
  }
  System.out.println ("");
}

public void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i, high);
    i = i + 1;
  }
}
```

Notice that when I added a new parameter, I had to change the first line of the method (the interface or prototype), and I also had to change the place where the method is invoked in `printMultTable`. As expected, this program generates a square 7x7 table:

```
1   2   3   4   5   6   7
2   4   6   8   10   12   14
3   6   9   12   15   18   21
4   8   12   16   20   24   28
5   10   15   20   25   30   35
6   12   18   24   30   36   42
7   14   21   28   35   42   49
```

When you generalize a method appropriately, you often find that the resulting program has capabilities you did not intend. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `printMultTable`. Change

```
    printMultiples (i, high);
```

to

```
    printMultiples (i, i);
```

and you get

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
```

I'll leave it up to you to figure out how it works.

7.8 Loops and lists

Many of the list operations in the previous chapter can also be written as loops. For example, to count the number of nodes in a list, we can traverse the list until we reach the empty list:

```
public int length (IntList list) {
   int count = 0;
   while (!isEmpty (list)) {
       count = count + 1;
       list = tail (list);
   }
   return count;
}
```

Each time through the list, the assignment `list = tail (list)` moves from one node to the next.

As an exercise, write a version of `sum` that uses a loop to add up the elements of an IntList.

7.9 Glossary

loop: A statement that executes repeatedly while or until some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

encapsulate: To divide a large complex program into components (like methods) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a method and that exists only within that method. Local variables cannot be accessed from outside their home method, and do not interfere with any other methods.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing.

Chapter 8

Objects as containers

8.1 Points and Rectangles

In this chapter, we are going to use two new object types that are part of the Java language, `Point` and `Rectangle`. Right from the start, I want to make it clear that these points and rectangles are not graphical objects that appear on the screen. They are variables that contain data, just like `int`s and `double`s. Like other variables, they are used internally to perform computations.

8.2 Packages

The built-in Java classes are divided into a number of **packages**, including `java.lang`, which the most commonly-used classes, and which is imported automatically, and `java.awt`, which contains classes that pertain to the Java **Abstract Window Toolkit** (AWT). The AWT contains classes for windows, buttons, graphics, etc.

In order to use a package, you have to **import** it using (surprise) an `import statment`. All `import` statements appear at the beginning of the program, outside the class definition. To import a specific class from a package:

```
import java.util.Stack;
```

This imports the `Stack` class from the `java.util` package. More often, we want to import all the classes from a package, using the `*` operator:

```
import java.awt.*;
```

This imports all the classes in the AWT package. The definitions of the `Point` and `Rectangle` classes are in `java.awt`, so any program that uses them should include this import statement.

The documentation for all Java packages is available online from

`http://java.sun.com/products/jdk/1.1/docs/api/packages.html`

for Java version 1.1 and

`http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html`

for Java version 1.2.

8.3 `Point` objects

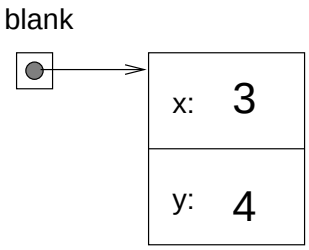
A point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

In Java, a point is represented by a `Point` object. To create a new point, you have to use the `new` command:

```
Point blank;
blank = new Point (3, 4);
```

The first line is a conventional variable declaration: `blank` has type `Point`. The second line invokes the `new` command to create the new point, $(3, 4)$.

The result of the `new` command is a **reference** to the new point, as shown here:



The big box shows the newly-created object with the instance variables for the Point class, `x` and `y`.

8.4 Instance variables

You can access the instance variables of an object using "dot notation."

```
int x = blank.x;
```

The expression `blank.x` means "go to the object `blank` refers to, and get the value of `x`." In this case we assign that value to a local variable named `x`. Notice that there is no conflict between the local variable named `x` and the instance variable named `x`. The purpose of dot notation is to identify *which* variable you are referring to unambiguously.

You can use dot notation as part of any Java expression, so the following are legal.

```
System.out.println (blank.x + ", " + blank.y);
int distance = blank.x * blank.x + blank.y * blank.y;
```

The first line prints `3, 4`; the second line calculates the value 25.

8.5 Objects as parameters

You can pass objects as parameters in the usual way. For example

```
public static void printPoint (Point p) {
    System.out.println ("(" + p.x + ", " + p.y + ")");
}
```

is a method that takes a point as an argument and prints it in the standard format. If you invoke `printPoint (blank)`, it will print `(3, 4)`. Actually, Java has a built-in method for printing `Points`. If you invoke `System.out.println (blank)`, you get

```
java.awt.Point[x=3,y=4]
```

The standard format for printing objects is the name of the type, followed by the contents of the object, including the names and values of the instance variables.

As a second example, we can write a `distance` method that takes two `Points` as parameters:

```
public static double distance (Point p1, Point p2) {
    double dx = (double)(p2.x - p1.x);
    double dy = (double)(p2.y - p1.y);
    return Math.sqrt (dx*dx + dy*dy);
}
```

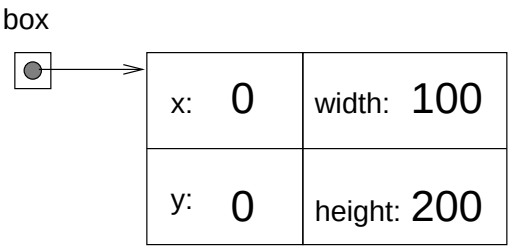
The typecasts are not really necessary; I just added them as a reminder that the instance variables in a `Point` are integers.

8.6 Rectangles

`Rectangle`s are similar to points, except that they have four instance variables, named `x`, `y`, `width` and `height`. Other than that, everything is pretty much the same.

```
Rectangle box = new Rectangle (0, 0, 100, 200);
```

creates a new `Rectangle` object and makes `box` refer to it. The figure shows the effect of this assignment.



If you print `box`, you get

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

Again, this is the result of a built-in Java method that knows how to print `Rectangle` objects.

8.7 Objects as return types

You can write methods that return objects. For example, `findCenter` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
public static Point findCenter (Rectangle box) {
    int x = box.x + box.width/2;
    int y = box.y + box.height/2;
    return new Point (x, y);
}
```

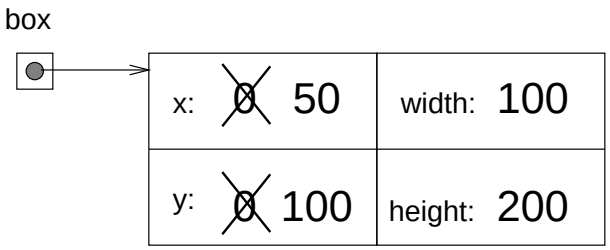
Notice that you can use `new` to create a new object, and then immediately use the result as a return value.

8.8 Objects are mutable

You can change the contents of an object by making an assignment to one of its instance variables. For example, to "move" a rectangle without changing its size, you could modify the `x` and `y` values:

```
box.x = box.x + 50;
box.y = box.y + 100;
```

The result is shown in the figure:



We could take this code and encapsulate it in a method, and generalize it to move the rectangle by any amount:

```
public static void moveRect (Rectangle box, int dx, int dy) {
    box.x = box.x + dx;
    box.y = box.y + dy;
}
```

The variables `dx` and `dy` indicate how far to move the rectangle in each direction. Invoking this method has the effect of modifying the `Rectangle` that is passed as an argument.

```
Rectangle box = new Rectangle (0, 0, 100, 200);
moveRect (box, 50, 100);
System.out.println (box);
```

prints `java.awt.Rectangle[x=50,y=100,width=100,height=200]`.

Modifying objects by passing them as arguments to methods can be useful, but it can also make debugging more difficult because it is not always clear which method invocations modify their arguments. Later, I will discuss some pros and cons of this programming style.

In the meantime, we can enjoy the luxury of Java's built-in methods, which include `translate`, which does exactly the same thing as `moveRect`, although the syntax for invoking it is a little different. It is an object method, so instead of passing the `Rectangle` as an argument, we invoke `translate` on the `Rectangle` and pass only `dx` and `dy` as arguments.

```
box.translate (50, 100);
```

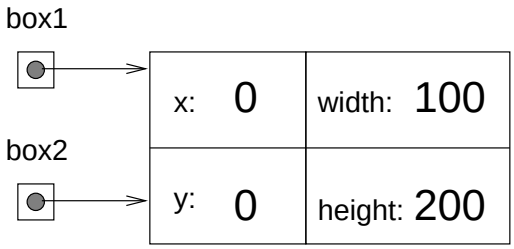
The effect is the same.

8.9 Aliasing

Remember that when you make an assignment to an object variable, you are assigning a *reference* to an object. It is possible to have multiple variables that refer to the same object. For example, this code:

```
Rectangle box1 = new Rectangle (0, 0, 100, 200);
Rectangle box2 = box1;
```

generates a state diagram that looks like this:

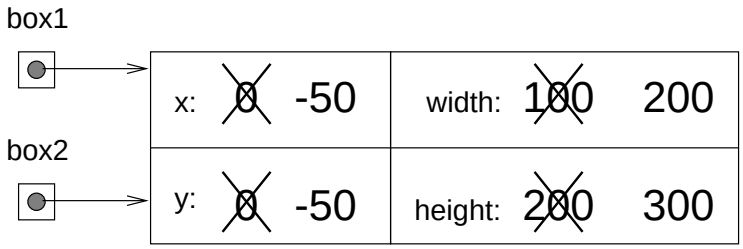


Both `box1` and `box2` refer or "point" to the same object. In other words, this object has two names, `box1` and `box2`. This is an example of aliasing.

When two variables are aliased, any changes that affect one variable also affect the other. For example:

```
System.out.println (box2.width);
box1.grow (50, 50);
System.out.println (box2.width);
```

The first line prints `100`, which is the width of the `Rectangle` referred to by `box2`. The second line invokes the `grow` method on `box1`, which expands the `Rectangle` by 50 pixels in every direction (see the documentation for more details). The effect is shown in the figure:



As should be clear from this figure, whatever changes are made to `box1` also apply to `box2`. Thus, the value printed by the third line is `200`, the width of the expanded rectangle. (As an aside, it is perfectly legal for the coordinates of a `Rectangle` to be negative.)

As you can tell even from this simple example, code that involves aliasing can get confusing fast, and it can be very difficult to debug. In general, aliasing should be avoided or used with care.

8.10 `null`

When you create an object variable, remember that you are creating a *reference* to an object. Until you make the variable point to an object, the value of the variable is `null`. `null` is a special value in Java (and a Java keyword) that is used to mean "no object."

The declaration `Point blank;` is equivalent to this initialization

```
Point blank = null;
```

and is shown in the following state diagram:

The value `null` is represented by a dot with no arrow.

If you try to use a null object, either by accessing an instance variable or invoking a method, you will get a `NullPointerException`. The system will print an error message and terminate the program.

```
Point blank = null;
int x = blank.x;              // NullPointerException
blank.translate (50, 50);     // NullPointerException
```

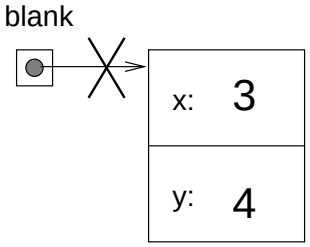
On the other hand, it is legal to pass a null object as an argument or receive one as a return value. In fact, it is common to do so, for example to represent an empty set or indicate an error condition.

8.11 Garbage collection

We have talked about what happens when more than one variable refers to the same object. What happens when *no* variable refers to an object? For example:

```
Point blank = new Point (3, 4);
blank = null;
```

The first line creates a new `Point` object and makes `blank` refer to it. The second line changes `blank` so that instead of referring to the object, it refers to nothing (the null object).



If no one refers to an object, then no one can read or write any of its values, or invoke a method on it. In effect, it ceases to exist. We could keep the object in memory, but it would only waste space, so periodically as your program runs, the Java system looks for stranded objects and reclaims them, in a process called

garbage collection. Later, the memory space occupied by the object will be available to be used as part of a new object.

You don't have to do anything to make garbage collection work, and in general you will not be aware of it.

8.12 Objects and primitives

There are two kinds of types in Java, primitive types and object types. Primitives, like `int` and `boolean` begin with lower-case letters; object types begin with upper-case letters. This distinction is useful because it reminds us of some of the differences between them:

- When you declare a primitive variable, you get storage space for a primitive value. When you declare an object variable, you get a space for a reference to an object. In order to get space for the object itself, you have to use the `new` command.
- If you don't initialize a primitive type, it is given a default value that depends on the type. For example, `0` for `int`s and `true` for `boolean`s. The default value for object types is `null`, which indicates no object.
- Primitive variables are well isolated in the sense that there is nothing you can do in one method that will affect a variable in another method. Object variables can be tricky to work with because they are not as well isolated. If you pass a reference to an object as an argument, the method you invoke might modify the object, in which case you will see the effect. The same is true when you invoke a method on an object. Of course, that can be a good thing, but you have to be aware of it.

There is one other difference between primitives and object types. You cannot add new primitives to the Java language (unless you get yourself on the standards committee), but you can create new object types! We'll see how in the next chapter.

8.13 Glossary

package: A collection of classes. The built-in Java classes are organized in packages.

AWT: The Abstract Window Toolkit, one of the biggest and most commonly-used Java packages.

instance: An example from a category. My cat is an instance of the category "feline things." Every object is an instance of some class.

instance variable: One of the named data items that make up an object. Each object (instance) has its own copy of the instance variables for its class.

reference: A value that indicates an object. In a state diagram, a reference appears as an arrow.

aliasing: The condition when two or more variables refer to the same object.

garbage collection: The process of finding objects that have no references and reclaiming their storage space.

primitive type: Types in Java that begin with a lower case letter are primitive types, which obey semantic rules that are different from object types.

Chapter 9

Arrays

An **array** is a compound data structure where each element is identified by an index. You can make an array of `int`s, `double`s, or any other type, but all the values in an array have to have the same type.

Syntactically, arrays types look like other Java types except they are followed by `[]`. For example, `int[]` is the type "array of integers" and `double[]` is the type "array of doubles."

You can declare variables with these types in the usual ways:

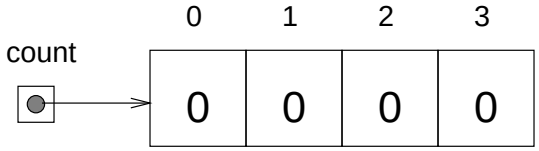
```
int[] count;
double[] values;
```

Until you initialize these variables, they are set to `null`. To create the array itself, use the `new` command.

```
count = new int[4];
values = new double[size];
```

The first assignment makes `count` refer to an array of 4 integers; the second makes `values` refer to an array of `double`s. The number of elements in `values` depends on `size`. You can use any integer expression as an array size.

The following figure shows how arrays are represented in state diagrams:



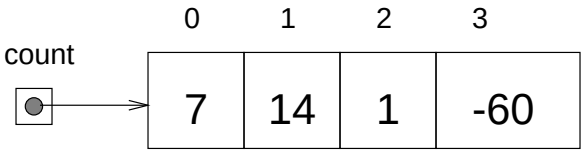
The large numbers inside the boxes are the **elements** of the array. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new array of integers, the elements are initialized to zero.

9.1 Accessing elements

To store values in the array, use the `[]` operator. For example `count[0]` refers to the "zeroeth" element of the array, and `count[1]` refers to the "oneth" element. You can use the `[]` operator anywhere in an expression:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

All of these are legal assignment statements. Here is the effect of this code fragment:



The indices of an array start with 0. For perverse reasons, computer scientists always start counting from zero. So, the four elements of this array are numbered from 0 to 3, which means that there is no element with the index 4.

It is a common error to go beyond the bounds of an array, which will cause an `ArrayOutOfBoundsException`. As with all exceptions, you get an error message and the program quits.

You can use any expression as an index, as long as it has type `int`. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;
while (i < 4) {
    System.out.println (count[i]);
    i++;
}
```

This is a standard `while` loop that counts from 0 up to 4, and when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

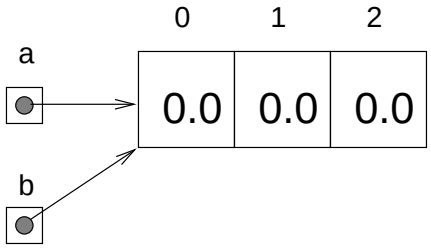
Each time through the loop we use `i` as an index into the array, printing the `i`th element. This type of array traversal is very common. Arrays and loops go together like fava beans and a nice Chianti.

9.2 Copying arrays

When you copy an array variable, remember that you are copying a reference to the array. For example:

```
double[] a = new double [3];
double[] b = a;
```

This code creates one array of three doubles, and sets two different variables to refer to it. This situation is a form of aliasing.



Any changes in either array will be reflected in the other. This is not usually the behavior you want; instead, you should make a copy of the array, by allocating a new array and copying each element from one to the other.

```
double[] b = new double [3];

int i = 0;
while (i < 4) {
  b[i] = a[i];
  i++;
}
```

9.3 for loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it.

This type of loop is so common that there is an alternate loop statement, called for, that expresses it more concisely. The general syntax looks like this:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
    BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while (CONDITION) {
    BODY
    INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
for (int i = 0; i < 4; i++) {
    System.out.println (count[i]);
}
```

is equivalent to

```
int i = 0;
while (i < 4) {
    System.out.println (count[i]);
    i++;
}
```

As an exercise, write a `for` loop to copy the elements of an array. As another exercise, write a `for` loop that traverses an IntList.

9.4 Arrays and objects

In many ways, arrays behave like objects:

- When you declare an array variable, you get a reference to an array.
- You have to use the `new` command to create the array itself.
- When you pass an array as an argument, you pass a reference, which means that the invoked method can change the contents of the array.

Some of the objects we have looked at, like `Rectangles`, are similar to arrays, in the sense that they are named collection of values. This raises the question, "How is an array of 4 integers different from a Rectangle object?"

If you go back to the definition of "array" at the beginning of the chapter, you will see one difference, which is that the elements of an array are identified by indices, whereas the elements (instance variables) of an object have names (like `x`, `width`, etc.).

Another difference between arrays and objects is that all the elements of an array have to be the same type. Although that is also true of `Rectangle`s, many objects have instance variables with different types,

9.5 Array length

Actually, arrays do have one named instance variable: `length`. Not surprisingly, it contains the length of the array (number of elements). It is a good idea to use this value as the upper bound of a loop, rather than a constant value. That way, if the size of the array changes, you won't have to go through the program changing all the loops; they will work correctly for any size array.

```
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

The last time the body of the loop gets executed, `i` is `a.length - 1`, which is the index of the last element. When `i` is equal to `a.length`, the condition fails and the body is not executed, which is a good thing, since it would cause an exception. This code assumes that the array `b` contains at least as many elements as `a`.

As an exercise, write a method called `cloneArray` that takes an array of integers as a parameter, creates a new array that is the same size, copies the elements from the first array into the new one, and then returns a reference to the new array.

9.6 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example, but there are many more.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. Java provides a built-in method that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes, they will do.

Check out the documentation of the `random` method in the `Math` class. The return value is a `double` between 0.0 and 1.0. Each time you invoke `random` you get a different randomly-generated number. To see a sample, run this loop:

```
for (int i = 0; i < 10; i++) {
    double x = Math.random ();
    System.out.println (x);
}
```

To generate a random `double` between 0.0 and an upper bound like `high`, you can multiply `x` by `high`. How would you generate a random number between `low` and `high`? How would you generate a random integer?

9.7 Two-Dimensional Arrays

So far we have only looked at one-dimensional arrays—a sequence of elements all of the same type. Well there's no reason those elements can't be arrays themselves. An array of arrays is a two-dimensional array.

The syntax for a two-dimensional array is similar to the syntax for a one dimensional array. For primitive types like integers, it looks like this:

```
int[][] array = new int[4][4];
```

The type of the variable `array` is `int[][]` which is pronounced "array of array of ints." This is not the same as the type `int[]` or the type `int`.

To assign one of the elements of the array, we use bracket operators.

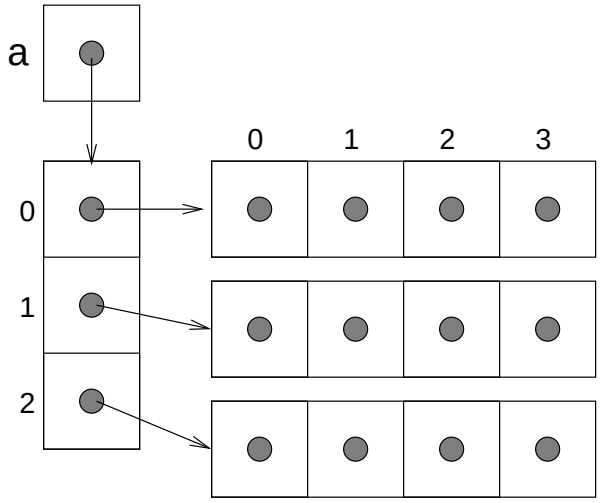
```
array[0][0] = 7;
```

This stores the value 7 in the first column of the first row of the array. All the other elements of the array are still zero.

Arrays of objects are similar. For example, `points` is an array of array of points.

```
Point[][] points = new Point[3][4];
```

Here is a graphical representation of `points`. Notice that the object pointers that make up the array are all `null`. That's because we have only allocated the array; we haven't created any `Point` objects yet.



To assign one of the elements of the array, we use bracket operators.

```
points[2][3] = new Point (4, 5);
```

This sets the lowest, rightmost element of the array to refer to a newly-minted Point object.

9.8 The Vector class

One of the problems with arrays is that their size is fixed. If you try to access something past the end of an array, it causes a run-time error. So it is up to the programmer to keep track of the size of the array and resize it when necessary.

Another problem is that all the elements of the array have to have the same type. We can solve both of these problems by replacing the array with a `Vector`.

The `Vector` is a built-in Java class in the `java.util` package. The elements of a Vector can be any kind of Object, and you can mix different objects within a single Vector. Also, Vectors resize themselves automatically as you add elements.

The `Vector` class provides methods named `get` and `set` that are equivalent to the bracket syntax we use to access the elements of an array. You should review the other `Vector` operations by consulting the online documentation.

Before using the `Vector` class, you should understand a few concepts. Every `Vector` has a capacity, which is the amount of space that has been allocated to store values, and a size, which is the number of values that are actually in the vector.

The following figure is a simple diagram of a `Vector` that contains three elements, but it has a capacity of seven.

In general, it is your responsibility to make sure that the vector has sufficient *size* before invoking `set` or `get`. If you try to access an element that does not exist (in this case the elements with indices 3 through 6), you will get an `ArrayIndexOutOfBounds` exception.

The `Vector` methods `add` and `insert` automatically increase the size of the `Vector`, but `set` does not. The `resize` method adds `null` elements to the end of the `Vector` to get to the given size.

Most of the time you don't have to worry about capacity. Whenever the size of the `Vector` changes, the capacity is updated automatically. For performance reasons, some applications might want to take control of this function, which is why there are additional methods for increasing and decreasing capacity.

Because we don't have access to the implementation of a vector, it is not clear how we should traverse one. Of course, one possibility is to use a loop variable as an index into the vector:

```
for (int i=0; i<v.size(); i++) {
    System.out.println (v.get(i));
}
```

There's nothing wrong with that, but there is another way that serves to demonstrate the `Iterator` class. Vectors provide a method named `iterator` that returns an `Iterator` object that makes it possible to traverse the vector.

9.9 The `Iterator` class

`Iterator` is an abstract class in the `java.util` package. It specifies three methods:

`hasNext`: Does this iteration have more elements?

`next`: Return the next element, or throw an exception if there is none.

`remove`: Remove from the collection the last element that was returned.

The following example uses an iterator to traverse and print the elements of a vector.

```
Iterator iterator = v.iterator ();
while (iterator.hasNext ()) {
    System.out.println (iterator.next ());
}
```

Once the `Iterator` is created, it is a separate object from the original `Vector`. Subsequent changes in the `Vector` are not reflected in the `Iterator`. In fact, if you modify the `Vector` after creating an `Iterator`, the `Iterator` becomes invalid. If you access the `Iterator` again, it will cause a `ConcurrentModification` exception.

Iterators make it possible to traverse a data structure without knowing the details of its implementation. In fact, there are other Java classes that provide Iterators; we can traverse any of them without even knowing what kind of object we have.

9.10 Glossary

array: A named collection of values, where all the values have the same type, and each value is identified by an index.

collection: Any data structure that contains a set of items or elements.

element: One of the values in an array. The `[]` operator selects elements of an array.

index: An integer variable or value used to indicate an element of an array.

deterministic: A program that does the same thing every time it is invoked.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

Chapter 10

Create your own objects

10.1 Class definitions and object types

In this chapter we will write class definitions that create new Java object types. The most important ideas in this chapter are

- Defining a new class also creates a new object type with the same name.
- A class definition is like a template for objects: it determines what instance variables the objects have and what methods can operate on them.
- Every object belongs to some object type; hence, it is an instance of some class.
- When you invoke the `new` command to create an object, Java invokes a special method called a **constructor** to initialize the instance variables. You provide one or more constructors as part of the class definition.
- Typically all the methods that operate on a type go in the class definition for that type.

Here are some syntax issues about class definitions:

- Class names (and hence object types) always begin with a capital letter, which helps distinguish them from primitive types and variable names.
- You usually put one class definition in each file, and the name of the file must be the same as the name of the class, with the suffix `.java`. For example, the `Time` class is defined in the file named `Time.java`.
- In any program, one class is designated as the **startup class**. The startup class must contain a method named `main`, which is where the execution of the program begins. Another class *may* have a method named `main`, but it will not be executed.

With those issues out of the way, let's look at an example of a user-defined type, `Time`.

10.2 Time

A common motivation for creating a new Object type is to take several related pieces of data and encapsulate them into an object that can be manipulated (passed as an argument, operated on) as a single unit. We have already seen two built-in types like this, `Point` and `Rectangle`.

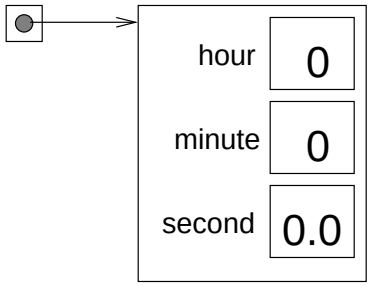
Another example, which we will implement ourselves, is `Time`, which is used to record the time of day. The various pieces of information that form a time are the hour, minute and second. Because every `Time` object will contain these data, we need to create instance variables to hold them.

The first step is to decide what type each variable should be. It seems clear that `hour` and `minute` should be integers. Just to keep things interesting, let's make `second` a `double`, so we can record fractions of a second.

Instance variables are declared at the beginning of the class definition, outside of any method definition, like this:

```
class Time {
    int hour, minute;
    double second;
}
```

All by itself, this code fragment is a legal class definition. The state diagram for a `Time` object would look like this:



After declaring the instance variables, the next step is usually to define a constructor for the new class.

10.3 Constructors

The usual role of a constructor is to initialize the instance variables. The syntax for constructors is similar to that of other methods, with three exceptions:

- The name of the constructor is the same as the name of the class.
- Constructors have no return type and no return value.
- The keyword `static` is omitted.

Here is an example for the `Time` class:

```
public Time () {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
}
```

Notice that where you would expect to see a return type, between `public` and `Time`, there is nothing. That's how we (and the compiler) can tell that this is a constructor.

This constructor does not take any arguments, as indicated by the empty parentheses `()`. Each line of the constructor initializes an instance variable to an arbitrary default value (in this case, midnight). The name `this` is a special keyword that is the name of the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods.

But you do not declare `this` and you do not use `new` to create it. In fact, you are not even allowed to make an assignment to it! `this` is created by the system; all you have to do is store values in its instance variables.

A common error when writing constructors is to put a `return` statement at the end. Resist the temptation.

10.4 More constructors

Constructors can be overloaded, just like other methods, which means that you can provide multiple constructors with different parameters. Java knows which constructor to invoke by matching the arguments of the `new` command with the parameters of the constructors.

It is very common to have one constructor that takes no arguments (shown above), and one constructor that takes a parameter list that is identical to the list of instance variables. For example:

```
public Time (int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
```

The names and types of the parameters are exactly the same as the names and types of the instance variables. All the constructor does is copy the information from the parameters to the instance variables.

If you go back and look at the documentation for `Point`s and `Rectangle`s, you will see that both classes provide constructors like this. Overloading constructors provides the flexibility to create an object first and then fill in the blanks, or to collect all the information before creating the object.

So far this might not seem very interesting, and in fact it is not. Writing constructors is a boring, mechanical process. Once you have written two, you will find that you can churn them out in your sleep, just by looking at the list of instance variables.

10.5 Creating a new object

Although constructors look like methods, you never invoke them directly. Instead, when you use the `new` command, the system allocates space for the new object and then invokes your constructor to initialize the instance variables.

The following program demonstrates two ways to create and initialize `Time` objects:

```
class Time {
    int hour, minute;
    double second;

    public Time () {
      this.hour = 0;
      this.minute = 0;
      this.second = 0.0;
    }

    public Time (int hour, int minute, double second) {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
    }

    public static void main (String[] args) {

      // one way to create and initialize a Time object
      Time t1 = new Time ();
      t1.hour = 11;
      t1.minute = 8;
      t1.second = 3.14159;
      System.out.println (t1);

      // another way to do the same thing
      Time t2 = new Time (11, 8, 3.14159);
      System.out.println (t2);
    }
}
```

As an exercise, figure out the flow of execution through this program.

In `main`, the first time we invoke the `new` command, we provide no arguments, so Java invokes the first constructor. The next few lines assign values to each of the instance variables.

The second time we invoke the `new` command, we provide arguments that match the parameters of the second constructor. This way of initializing the instance variables is more concise (and slightly more efficient), but it can be harder to read, since it is not as clear which values are assigned to which instance variables.

10.6 Printing an object

The output of the previous program is:

```
Time@80cc7c0
Time@80cc807
```

When Java prints the value of a user-defined object type, it prints the name of the type and a special hexadecimal (base 16) code that is unique for each object. This code is not meaningful in itself; in fact, it can vary from machine to machine and even from run to run. But it can be useful for debugging, in case you want to keep track of individual objects.

In order to print objects in a way that is more meaningful to users you should provide a method named `toString` that returns a String representation of the object.

When you print an object using `print` or `println`, Java checks to see whether you have provided `toString`, and if so it invokes it. If not, it invokes a default version of `toString` that produces the output we just saw.

```
public String toString () {
    return hour + ":" + minute + ":" + second;
}
```

The output of this method, if we print either `t1` or `t2`, is `11:8:3.14159`. Although this is recognizable as a time, it is not quite in the standard format. For example, if the number of minutes or seconds is less than 10, we expect a leading `0` as a place-keeper. Also, we might want to drop the decimal part of the seconds. In other words, we want something like `11:08:03`.

In most languages, there are simple ways to control the output format for numbers. In Java there are no simple ways.

Java provides very powerful tools for printing formatted things like times and dates, and also for interpreting formatted input. Unfortunately, these tools are not very easy to use, so I am going to leave them out of this book. If you want, though, you can take a look at the documentation for the `Date` class in the `java.util` package.

10.7 Operations on objects

Even though we can't print times in an optimal format, we can still write methods that manipulate `Time` objects. In the next few sections, I will demonstrate several of the possible interfaces for methods that operate on objects. For some operations, you will have a choice of several possible interfaces, so you should consider the pros and cons of each:

pure function: Takes objects and/or primitives as arguments but does not modify the objects. The return value is either a primitive or a new object created inside the method.

modifier: Takes objects as arguments and modifies some or all of them. Often returns void.

10.8 Pure functions

A method is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or printing something. The only result of invoking a pure function is the return value.

One example is `after`, which compares the current time, `this`, to another time that is provided as a parameter. It returns a `boolean` that indicates whether the current time comes after the parameter:

```
public boolean after (Time t2) {
    if (hour > t2.hour) return true;
    if (hour < t2.hour) return false;

    if (minute > t2.minute) return true;
    if (minute < t2.minute) return false;

    if (second > t2.second) return true;
    return false;
}
```

What is the result of this method if the two times are equal? Does that seem like the appropriate result for this method? If you were writing the documentation for this method, would you mention that case specifically?

A second example is `add`, which calculates the sum of the current time and another time provided as a parmeter. For example, if it is `9:14:30`, and your breadmaker takes 3 hours and 35 minutes, you could use `addTime` to figure out when the bread will be done.

Here is a rough draft of this method that is not quite right:

```
public Time add (Time t2) {
    Time sum = new Time ();
    sum.hour = hour + t2.hour;
```

```
    sum.minute = minute + t2.minute;
    sum.second = second + t2.second;
    return sum;
}
```

Although this method returns a `Time` object, it is not a constructor. You should go back and compare the syntax of a method like this with the syntax of a constructor, because it is easy to get confused.

Here is an example of how to use this method. If `currentTime` contains the current time and `breadTime` contains the amount of time it takes for your breadmaker to make bread, then you could use `addTime` to figure out when the bread will be done.

```
    Time currentTime = new Time (9, 14, 30.0);
    Time breadTime = new Time (3, 35, 0.0);
    Time doneTime = currentTime.add (breadTime);
    printTime (doneTime);
```

The output of this program is `12:49:30.0`, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this method does not deal with cases where the number of seconds or minutes adds up to more than 60. In that case, we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a second, corrected version of this method.

```
public Time add (Time t2) {
    Time sum = new Time ();
    sum.hour = hour + t2.hour;
    sum.minute = minute + t2.minute;
    sum.second = second + t2.second;

    if (sum.second >= 60.0) {
        sum.second -= 60.0;
        sum.minute += 1;
    }
    if (sum.minute >= 60) {
        sum.minute -= 60;
        sum.hour += 1;
    }
    return sum;
}
```

Although this version is correct, it's starting to get big. Later, I will suggest an alternate approach to this problem that will be much shorter.

This code demonstrates two operators we have not seen before, `+=` and `-=`. These operators provide a concise way to increment and decrement variables.

They are similar to `++` and `--`, except (1) they work on `double`s as well as `int`s, and (2) the amount of the increment does not have to be 1. The statement `sum.second -= 60.0;` is equivalent to `sum.second = sum.second - 60;`

10.9 Modifiers

As an example of a modifier, consider `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this method looks like:

```
public void increment (double secs) {
    second += secs;

    if (second >= 60.0) {
        second -= 60.0;
        minute += 1;
    }
    if (minute >= 60) {
        minute -= 60;
        hour += 1;
    }
}
```

The first line performs the basic operation; the remainder deals with the same cases we saw before.

Is this method correct? What happens if the argument `secs` is much greater than 60? In that case, it is not enough to subtract 60 once; we have to keep doing it until `second` is below 60. We can do that by simply replacing the `if` statements with `while` statements:

```
public void increment (double secs) {
    second += secs;

    while (second >= 60.0) {
        second -= 60.0;
        minute += 1;
    }
    while (minute >= 60) {
        minute -= 60;
        hour += 1;
    }
}
```

This solution is correct, but not very efficient. Can you think of a solution that does not require iteration?

10.10 Which is best?

Anything that can be done with modifiers can also be done with pure functions. In fact, there are programming languages, called **functional** programming languages, that only allow pure functions. Some programmers believe that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, there are times when modifiers are convenient, and some cases where functional programs are less efficient.

In general, I recommend that you write pure functions whenever it is reasonable to do so, and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

10.11 Incremental development vs. planning

In this chapter I have demonstrated an approach to program development I refer to as **rapid prototyping with iterative improvement**. In each case, I wrote a rough draft (or prototype) that performed the basic calculation, and then tested it on a few cases, correcting flaws as I found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to convince yourself that you have found *all* the errors.

An alternative is high-level planning, in which a little insight into the problem can make the programming much easier. In this case the insight is that a `Time` is really a three-digit number in base 60! The `second` is the "ones column," the `minute` is the "60's column", and the `hour` is the "3600's column."

When we wrote `add` and `increment`, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Thus an alternate approach to the whole problem is to convert `Time`s into `double`s and take advantage of the fact that the computer already knows how to do arithmetic with `double`s. Here is a method that converts a `Time` into a `double`:

```
public double convertToSeconds () {
    int minutes = hour * 60 + minute;
    double seconds = minutes * 60 + second;
    return seconds;
}
```

Now all we need is a way to convert from a `double` to a `Time` object. We could write a method to do it, but it might make more sense to write it as a third constructor:

```
public Time (double secs) {
    this.hour = (int) (secs / 3600.0);
    secs -= this.hour * 3600.0;
    this.minute = (int) (secs / 60.0);
```

```
    secs -= this.minute * 60;
    this.second = secs;
}
```

This constructor is a little different from the others, since it involves some calculation along with assignments to the instance variables.

You might have to think a bit to convince yourself that the technique I am using to convert from one base to another is correct. Assuming you are convinced, we can use these methods to rewrite `add`:

```
public Time add (Time t2) {
    double seconds = this.convertToSeconds () + t2.convertToSeconds ();
    return new Time (seconds);
}
```

This is much shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the methods it invokes are correct). As an exercise, rewrite `increment` the same way.

10.12 Generalization

In some ways converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers, and make the investment of writing the conversion methods (`convertToSeconds` and the third constructor), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add more features later. For example, imagine subtracting two `Time`s to find the duration between them. The naive approach would be to implement subtraction complete with "borrowing." Using the conversion methods would be much easier.

Ironically, sometimes making a problem harder (more general) makes is easier (fewer special cases, fewer opportunities for error).

10.13 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. I mentioned this word in Chapter 1, but did not define it carefully. It is not easy to define, so I will try a couple of approaches.

First, consider some things that are not algorithms. For example, when you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions, so that knowledge is not really algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the most difficult to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

Later you will have the opportunity to design simple algorithms for a variety of problems.

10.14 Glossary

class: Previously, I defined a class as a collection of related methods. In this chapter we learned that a class definition is also a template for a new type of object.

instance: A member of a class. Every object is an instance of some class.

constructor: A special method that initializes the instance variables of a newly-constructed object.

project: A collection of one or more class definitions (one per file) that make up a program.

startup class: The class that contains the `main` method where execution of the program begins.

function: A method whose result depends only on its parameters, and that has no side-effects other than returning a value.

functional programming style: A style of program design in which the majority of methods are functions.

modifier: A method that changes one or more of the objects it receives as parameters, and usually returns `void`.

algorithm: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

Chapter 11

Data abstraction and Graphics

11.1 The BankAccount class

Suppose we want to create a class of objects to represent bank accounts.

The first question to ask is what information to store in the BankAccount object. In other words, what instance variables should an account have?

As a simple example, we will define a BankAccount object with a String for the customer's name and three doubles for the balance in the checking and savings accounts and the total balance.

The next question to ask is what operations we might want for this class. Here are some basic operations that are pretty likely.

Create an account: initialize the instance variables.

Make a deposit: add money to one of the accounts.

Make a withdrawal: deduct money from one of the accounts.

Get a balance: return the current balance of one of the accounts, or the total.

Get the name: return the bank account owner's name.

These operations guide our decisions about what methods the BankAccount object will support.

11.1.1 The class definition

Since we know what the instance variables are, the initial class definition is straightforward.

```
class BankAccount {
    private String name;
    private double checking;
    private double savings;
    private double total;
}
```

All the instance variables are private, which means that they cannot be accessed from outside this class definition. From any other class, the only way to access these variables is through the methods we will provide.

We'll see in a minute why that might be a good thing.

11.1.2 Constructors

Once you have written the instance variables, writing a constructor is pretty much a mechanical process.

```
    public BankAccount (String name, double checking, double savings) {
        this.name = name;
        this.checking = checking;
        this.savings = savings;
        this.total = checking + savings;
    }
```

As usual, the list of parameters looks a lot like the list of instance variables, and most of what the constructor does is copy the values provided as arguments into the object. The only interesting thing is that we have to compute the total.

11.1.3 Accessor Methods

Since the instance variables are private, we have to provide methods that allow other classes to obtain the values we want them to obtain. These methods are called **accessor methods**. First we will write the "get" methods. Generating them is pretty much a mechanical process.

```
    public String getName () {
        return name;
    }

    public double getChecking () {
        return checking;
    }

    public double getSavings () {
        return savings;
    }
```

```
public double getTotal () {
    return total;
}
```

You might be wondering why we bothered making these variables private if we were planning to let people read them anyway.

The answer is that accessor methods make it possible to restrict operations that should not be performed. For example, we probably want to make it impossible to change the name on the account. Also, we shouldn't let anyone change `total` directly; instead, `total` should get updated whenever `checking` or `savings` gets updated. Finally, we want to allow deposits and withdrawals, but we might not want to make it possible to set the balance of the accounts directly.

So the only methods we'll provide for changing the instance variables are `updateChecking` and `updateSavings`:

```
public void updateChecking (double amount) {
    checking += amount;
    total += amount;
}

public void updateSavings (double amount) {
    savings += amount;
    total += amount;
}
```

You might have been expecting different methods for deposits and withdrawals, but that's unnecessary. A withdrawal is just a deposit with a negative amount.

One of the advantages of private variables and accessor methods is that we can choose which operations to allow and which to forbid.

Another advantage is that we can maintain the **consistency** of objects. In general, the definition of consistency depends on what kind of objects you are talking about. For a bank account, consistency means that the value of `total` should be the sum of `checking` and `savings`. If it is not, something has gone horribly wrong, and it is likely to cause a problem at some point (like when the auditors come).

In this case we can prove that `total` can never be wrong, because it was correct when we constructed the object, and all the update methods update `total` correctly. Since no other classes can access these variables, there is no way for them to mess up the system. If the instance variables were public, we would have to find every place in the program that updates BankAccounts to be sure that they maintain consistency.

A third advantage is that we can add code to the accessor methods to check for errors or special conditions. In the example, we might want the update commands to check for overdrafts:

```
public void updateSavings (double amount) {
```

```
if (savings + amount < 0) {
            handleOverdraft ();
            return;
        }
        savings += amount;
        total += amount;
    }
```

If performing the update would make the balance negative, we invoke an imaginary method that handles overdrafts and then return without performing the update.

11.2 Data Abstraction

The final advantage of this style of programming is that it provides **data abstraction**, which means that the other classes that use these methods don't know anything about how the data is represented.

This abstraction makes it possible to change the details of the implementation without affecting other parts of the program. For example, if it turns out that bank accounts take up too much space (assuming there are a lot of them), we might want to save a few bytes by getting rid of the instance variable `total`.

All we have to do is delete `total` from the program everywhere it appears (which actually makes the update methods smaller and faster), and then change the implementation of `getTotal`:

```
    public double getTotal () {
        return checking + savings;
    }
```

The new version of the program is smaller, more likely to be correct (since we don't have to worry about consistency), and probably faster, since we only have to compute the total if someone asks, not every time there is an update.

Even more importantly, because of data abstraction, we could switch from one representation to the other and no one would ever know! The contract for the BankAccount class never changed. This flexibility is particularly important for large software projects where different programmers might be working on different parts of the program. Data abstraction allows these groups to work independently without creating problems for each other.

11.3 Applets

We have been writing and modifying Java Applets all semester, but we haven't seen much about how they work.

The Applet class is a build-in Java class that defines the basic operations that all Applets perform. It is defined in the `java.applet` package.

To create a new Applet, you have to import `java.applet.Applet` and create a new class that extends `Applet`.

```
import java.awt.*;
import java.applet.Applet;

public class GraphicsWorld extends Applet {
}
```

In this case we also import the AWT, so we can use the methods in the `Graphics` class.

The Applet class defines four methods that are invoked automatically when the Applet runs:

`init`: invoked when the applet is loaded, `init` performs any initialization the applet requires.

`start`: invoked when the applet starts running.

`stop`: invoked whenever the applet stops, usually when the window closes.

`paint`: invoked when the applet starts, and then invoked again whenever the applet needs to redraw the window.

We define the behavior of a new applet by overriding one or more of these methods.

11.4 The `paint` method

For the examples we will look at, there's no need to override `init`, `start` or `stop`. We'll stick with the default implementation, which does nothing.

But we will override `paint`. The `paint` method we provide has to take a Graphics object as a parameter and return void.

```
public void paint (Graphics g) {
    System.out.println ("paint invoked");
}
```

In this case, all the method does is display a message in the console to indicate that it has been invoked.

The Graphics object you get as a parameter makes it possible to draw basic two-dimensional shapes in your Applet.

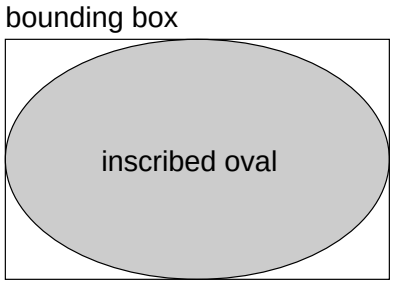
11.5 Drawing

To draw things on the screen, you invoke methods on the graphics object.

```
g.setColor (Color.black);
g.drawOval (x, y, width, height);
```

`setColor` changes the current color, in this case to black. Everything that gets drawn will be black, until we use `setColor` again.

`drawOval` takes four integers as arguments. These arguments specify a **bounding box**, which is the rectangle in which the oval will be drawn (as shown in the figure). The bounding box itself is not drawn; only the oval is. The bounding box is like a guideline. Bounding boxes are always oriented horizontally or vertically; they are never at a funny angle.

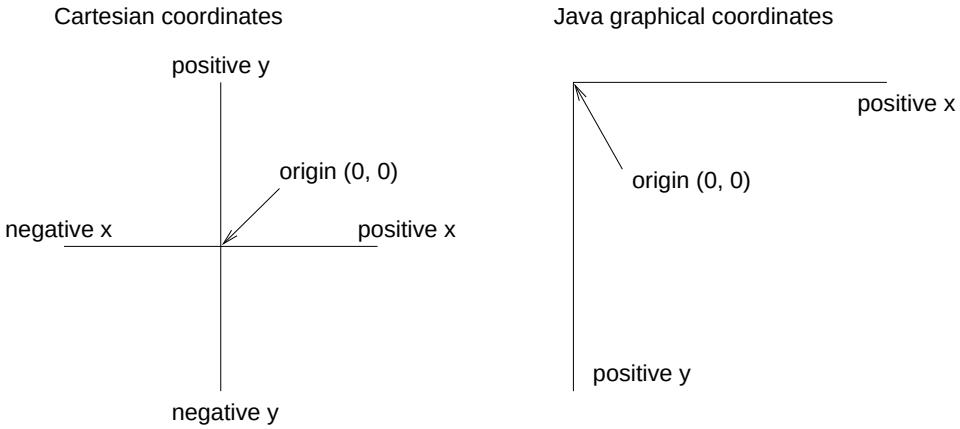


If you think about it, there are lots of ways to specify the location and size of a rectangle. You could give the location of the center or any of the corners, along with the height and width. Or, you could give the location of opposing corners. The choice is arbitrary, but in any case it will require the same number of parameters: four.

By convention, the usual way to specify a bounding box is to give the location of the *upper-left* corner and the width and height. The usual way to specify a location is to use a **coordinate system**.

11.6 Coordinates

You are probably familiar with Cartesian coordinates in two dimensions, in which each location is identified by an x-coordinate (distance along the x-axis) and a y-coordinate. By convention, Cartesian coordinates increase to the right and up, as shown in the figure.



Annoyingly, it is conventional for computer graphics systems to use a variation on Cartesian coordinates in which the origin is in the upper-left corner of the screen or window, and the direction of the positive y-axis is *down*. Java follows this convention.

The unit of measure is called a **pixel**; a typical screen is about 1000 pixels wide. Coordinates are always integers. If you want to use a floating-point value as a coordinate, you have to round it off to an integer.

11.7 A lame Mickey Mouse

Let's say we want to draw a picture of Mickey Mouse. We can use the oval we just drew as the face, and then add ears. Before we do that it is a good idea to break the program up into two methods. `paint` finds the dimensions of the Applet window and `draw` does the actual drawing.

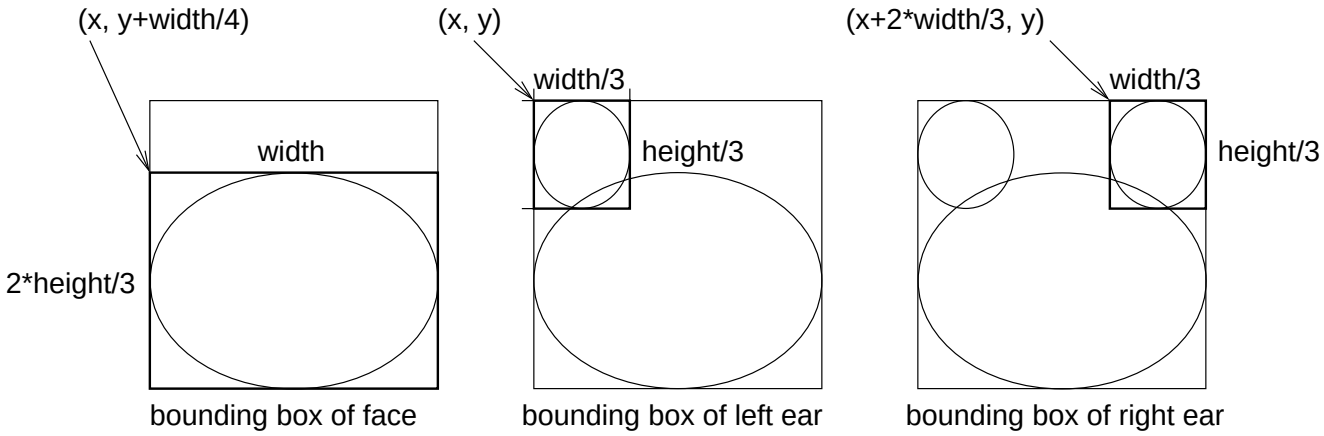
```
public void paint (Graphics g) {
    setBackground (Color.white);
    Rectangle r = g.getClipBounds ();

    draw (g, r.x, r.y, r.width, r.height);
}

public void draw (Graphics g, int x, int y, int width, int height) {
    g.drawOval (x, y+height/4, width, 2*height/3);
    g.drawOval (x, y, width/3, height/3);
    g.drawOval (x+2*width/3, y, width/3, height/3);
}
```

`setBackground` sets the background color to white. `getClipBounds` gets a Rectangle from the Graphics object that indicates the position and size of the region we can draw on. I doesn't cause an error if we draw outside the bounds, but the excess gets clipped.

The parameters for `draw` are the `Graphics` object and a bounding box. `draw` invokes `drawOval` three times, to draw Mickey's face and two ears. The following figure shows the bounding boxes for the ears.



The coordinates of the bounding boxes are all relative to the location (`x` and `y`) and size (`width` and `height`) of the original bounding box. As a result, we can use `draw` to draw a Mickey Mouse (albeit a lame one) anywhere on the screen in any size. As an exercise, modify the arguments passed to `draw` so that Mickey is one half the height and width of the screen, and centered.

11.8 Other drawing commands

Other drawing commands include

```
drawRect (int x, int y, int width, int height)
```

which draws a rectangle with the given bounding box, and

```
drawLine (int x1, int y1, int x2, int y2)
```

which draws a line from the point `(x1, y1)` to the point `(x2, y2)`.

One other command you might want to try is

```
drawRoundRect (int x, int y, int width, int height,
       int arcWidth, int arcHeight)
```

The first four parameters specify the bounding box of the rectangle; the remaining two parameters indicate how rounded the corners should be, specifying the horizontal and vertical diameter of the arcs at the corners.

There are also "fill" versions of these commands, that not only draw the outline of a shape, but also fill it in. The interfaces are identical; only the names have been changed:

```
fillOval (int x, int y, int width, int height)
fillRect (int x, int y, int width, int height)
fillRoundRect (int x, int y, int width, int height,
       int arcWidth, int arcHeight)
```

There is no such thing as `fillLine`—it just doesn't make sense.

11.9 A fractal Mickey Mouse

If you write recursive graphical methods, you often get interesting shapes called **fractals**. To draw a simple fractal we can write a version of `draw` so that it calls itself recursively:

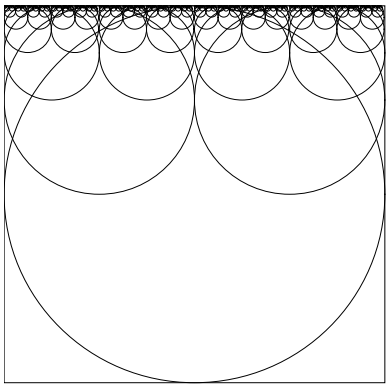
```
public void draw (Graphics g, int x, int y, int width, int height) {
    if (height == 0) return;

    g.drawOval (x, y, width, height);
    draw (g, x, y, width/2, height/2);
    draw (g, x+width/2, y, width/2, height/2);
}
```

The first line is the base case of the recursion. It checks whether the height is equal to zero, and if it is, it returns immediately without drawing any circles, and without making any recursive calls. Otherwise we would just draw smaller and smaller circles and eventually get a StackOverflowException.

The recursive step draws one oval and makes two recursive invocations. The first draw command draws a circle the size of the given bounding box. The two recursive calls use bounding boxes that are half the size of the original. One is in the upper left corner; the other is in the upper right.

The output of this program looks like this:



11.10 Glossary

accessor method: A method provided to get or set the value of an instance variable, usually from another class definition.

consistency: A property of an object that should be maintained as operations update the state of the object.

data abstraction: A form of modularity in which the users of a class don't see the implementation details of the class, especially the instance variables.

bounding box: An abstract rectangle (not drawn) that represents the boundaries of a graphical figure.

coordinate system: A way of denoting points in space relative to an origin.

pixel: A "picture element," used as a unit of measure for graphical operations.

fractal: An image that is defined recursively, so that part of the picture looks like a scaled down version of the whole.

Chapter 12

Strings and things

There are quite a few methods the operate on `String`s, documented in the String API.

The first method we will look at here is `charAt`, which allows you to extract letters from a `String`. In order to store the result, we need a variable type that can store individual letters (as opposed to strings). Individual letters are called characters, and the variable type that stores them is called `char`.

`char`s work just like the other types we have seen:

```
char fred = 'c';
if (fred == 'c') {
  System.out.println (fred);
}
```

Character values appear in single quotes (`'c'`). Unlike string values (which appear in double quotes), character values can contain only a single letter.

Here's how the `charAt` method is used:

```
String fruit = "banana";
char letter = fruit.charAt(1);
System.out.println (letter);
```

The syntax `fruit.charAt` indicates that I am invoking the `charAt` method on the object named `fruit`. I am passing the argument `1` to this method, which indicates that I would like to know the first letter of the string. The result is a character, which is stored in a `char` named `letter`. When I print the value of `letter`, I get a surprise:

```
a
```

`a` is not the first letter of `"banana"`. Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zeroeth") of `"banana"` is `b`. The 1th letter ("oneth") is `a` and the 2th ("twoeth") letter is `n`.

If you want the the zereoth letter of a string, you have to pass zero as an argument:

```
char letter = fruit.charAt(0);
```

12.1 Length

The second `String` method we'll look at is `length`, which returns the number of characters in the string. For example:

```
int length = fruit.length();
```

`length` takes no arguments,as indicated by `()`, and returns an integer, in this case 6. Notice that it is legal to have a variable with the same name as a method (although it can be confusing for human readers).

To find the last letter of a string, you might be tempted to try something like

```
int length = fruit.length();
char last = fruit.charAt (length);       // WRONG!!
```

That won't work. The reason is that there is no 6th letter in `"banana"`. Since we started counting at 0, the 6 letters are numbered from 0 to 5. To get the last character, you have to subtract 1 from `length`.

```
int length = fruit.length();
char last = fruit.charAt (length-1);
```

12.2 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a `while` statement:

```
int index = 0;
while (index < fruit.length()) {
  char letter = fruit.charAt (index);
  System.out.println (letter);
  index = index + 1;
}
```

This loop traverses the string and prints each letter on a line by itself. Notice that the condition is `index < fruit.length()`, which means that when `index` is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `fruit.length()-1`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set (in this case the set of characters in the string). The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a method that takes a `String` as an argument and that prints the letters backwards, all on one line.

12.3 Run-time errors

Way back in Section 1.4.2 I talked about run-time errors, which are errors that don't appear until a program has started running. In Java run-time errors are called **exceptions**.

If you use the `charAt` command and you provide an index that is negative or greater than `length-1`, you will get an exception: specifically, a `StringIndexOutOfBoundsException`. Try it and see how it looks.

If your program causes an exception, it prints an error message indicating the type of exception and where in the program it occurred. Then the program terminates.

12.4 Reading documentation

The documentation for `charAt` looks like this:

```
public char charAt(int index)

Returns the character at the specified index.
An index ranges from 0 to length() - 1.

Parameters: index - the index of the character.

Returns: the character at the specified index of this string.
         The first character is at index 0.

Throws: StringIndexOutOfBoundsException if the index is out of range.
```

The first line is the method's `prototype` (see Section 11.8), which indicates the name of the method, the type of the parameters, and the return type.

The next line describes what the method does. The next two lines explain the parameters and return values. In this case the explanations are a bit redundant, but the documentation is supposed to fit a standard format. The last line explains what exceptions, if any, can be caused by this method.

12.5 The `indexOf` method

In some ways, `indexOf` is the opposite of `charAt`. `charAt` takes an index and returns the character at that index. `indexOf` takes a character and finds the index where that character appears.

`charAt` fails if the index is out of range, and causes an exception. `indexOf` fails if the character does not appear in the string, and returns the value `-1`.

```
String fruit = "banana";
int index = fruit.indexOf('a');
```

This finds the index of the letter `'a'` in the string. In this case, the letter appears three times, so it is not obvious what `indexOf` should do. According to the documentation, it returns the index of the *first* appearance.

In order to find subsequent appearances, there is an alternate version of `indexOf` (for an explanation of this kind of overloading, see Section 10.4). It takes a second argument that indicates where in the string to start looking. If we invoke

```
int index = fruit.indexOf('a', 2);
```

it will start at the twoeth letter (the first `n`) and find the second `a`, which is at index 3. If the letter happens to appear at the starting index, the starting index is the answer. Thus,

```
int index = fruit.indexOf('a', 5);
```

returns 5. Based on the documentation, it is a little tricky to figure out what happens if the starting index is out of range:

> `indexOf` returns the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur.

One way to figure out what this means is to try out a couple of cases. Here are the results of my experiments:

- If the starting index is greater than or equal to `length()`, the result is -1, indicating that the letter does not appear at any index greater than the starting index.
- If the starting index is negative, the result is 1, indicating the first appearance of the letter at an index greater than the starting index.

If you go back and look at the documentation, you'll see that this behavior is consistent with the definition, even if it was not immediately obvious. Now that we have a better idea how `indexOf` works, we can use it as part of a program.

12.6 Looping and counting

The following program counts the number of times the letter `'a'` appears in a string:

```
String fruit = "banana";
int length = fruit.length();
int count = 0;

int index = 0;
while (index < length) {
  if (fruit.charAt(index) == 'a') {
    count = count + 1;
  }
  index = index + 1;
}
System.out.println (count);
```

This program demonstrates a common idiom, called a **counter**. The variable `count` is initialized to zero and then incremented each time we find an `'a'` (to **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to **excrement**, which is a noun). When we exit the loop, `count` contains the result: the total number of a's.

As an exercise, encapsulate this code in a method named `countLetters`, and generalize it so that it accepts the string and the letter as arguments.

As a second exercise, rewrite the method so that it uses `indexOf` to locate the a's, rather than checking the characters one by one.

12.7 Increment and decrement operators

Incrementing and decrementing are such common operations that Java provides special operators for them. The `++` operator adds one to the current value of an `int` or `char`. `--` subtracts one. Neither operator works on `double`s, `boolean`s or `String`s.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

```
System.out.println (i++);
```

Looking at this, it is not clear whether the increment will take effect before or after the value is printed. Because expressions like this tend to be confusing, I would discourage you from using them. In fact, to discourage you even more, I'm not going to tell you what the result is. If you really want to know, you can try it.

Using the increment operators, we can rewrite the letter-counter:

```
int index = 0;
while (index < length) {
  if (fruit.charAt(index) == 'a') {
    count++;
  }
  index++;
}
```

It is a common error to write something like

```
index = index++;              // WRONG!!
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of `index` unchanged. This is often a difficult bug to track down.

Remember, you can write `index = index +1;`, or you can write `index++;`, but you shouldn't mix them.

12.8 Character arithmetic

It may seem odd, but you can do arithmetic with characters! The expression `'a' + 1` yields the value `'b'`. Similarly, if you have a variable named `letter` that contains a character, then `letter - 'a'` will tell you where in the alphabet it appears (keeping in mind that 'a' is the zeroeth letter of the alphabet and 'z' is the 25th).

This sort of thing is useful for converting between the characters that contain numbers, like '0', '1' and '2', and the corresponding integers. They are not the same thing. For example, if you try this

```
char letter = '3';
int x = (int) letter;
System.out.println (x);
```

you might expect the value 3, but depending on your environment, you might get 51, which is the ASCII code that is used to represent the character '3', or you might get something else altogether. To convert '3' to the corresponding integer value you can subtract '0':

```
int x = (int)(letter - '0');
```

Technically, in both of these examples the typecast (`(int)`) is unnecessary, since Java will convert type `char` to type `int` automatically. I included the typecasts to emphasize the difference between the types, and because I'm a stickler about that sort of thing.

Since this conversion can be a little ugly, it is preferable to use the `digit` method in the `Character` class. For example:

```
int x = Character.digit (letter, 10);
```

converts `letter` to the corresponding digit, interpreting it as a base 10 number.

Another use for character arithmetic is to loop through the letters of the alphabet in order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings form an abecedarian series: Jack, Kack, Lack, Mack, Nack, Ouack, Pack and Quack. Here is a loop that prints these names in order:

```
char letter = 'J';
while (letter <= 'Q') {
  System.out.println (letter + "ack");
  letter++;
}
```

Notice that in addition to the arithmetic operators, we can also use the conditional operators on characters. The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because I've misspelled "Ouack" and "Quack." As an exercise, modify the program to correct this error.

12.8.1 Typecasting for experts

Here's a puzzler: normally, the statement `x++` is exactly equivalent to `x = x + 1`. Unless `x` is a `char`! In that case, `x++` is legal, but `x = x + 1` causes an error.

Try it out and see what the error message is, then see if you can figure out what is going on.

12.9 Strings are immutable

As you look over the documentation of the `String` methods, you might notice `toUpperCase` and `toLowerCase`. These methods are often a source of confusion, because it sounds like they have the effect of changing (or mutating) an existing string. Actually, neither these methods nor any others can change a string, because strings are **immutable**.

When you invoke `toUpperCase` on a `String`, you get a *new* `String` as a return value. For example:

```
String name = "Alan Turing";
String upperName = name.toUpperCase ();
```

After the second line is executed, `upperName` contains the value `"ALAN TURING"`, but `name` still contains `"Alan Turing"`.

12.10 `Strings` are incomparable

It is often necessary to compare strings to see if they are the same, or to see which comes first in alphabetical order. It would be nice if we could use the comparison operators, like `==` and `>`, but we can't.

In order to compare `String`s, we have to use the `equals` and `compareTo` methods. For example:

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";

if (name1.equals (name2)) {
  System.out.println ("The names are the same.");
}

int flag = name1.compareTo (name2);
if (flag == 0) {
  System.out.println ("The names are the same.");
} else if (flag < 0) {
  System.out.println ("name1 comes before name2.");
} else if (flag > 0) {
  System.out.println ("name2 comes before name1.");
}
```

The syntax here is a little weird. To compare two things, you have to invoke a method on one of them and pass the other as an argument.

The return value from `equals` is straightforward enough; `true` if the strings contain the same characters, and `false` otherwise.

The return value from `compareTo` is a little odd. It is the difference between the first characters in the strings that differ. If the strings are equal, it is 0. If the first string (the one on which the method is invoked) comes first in the alphabet, the difference is negative. Otherwise, the difference is positive. In this case the return value is positive 8, because the second letter of "Ada" comes before the second letter of "Alan" by 8 letters.

Using `compareTo` is often tricky, and I never remember which way is which without looking it up, but the good news is that the interface is pretty standard for comparing many types of objects, so once you get it you are all set.

Just for completeness, I should admit that it is *legal*, but very seldom *correct*, to use the `==` operator with `String`s. But what that means will not make sense until later, so for now, don't do it.

12.11 Glossary

object: A collection of related data that comes with a set of methods that operate on it.

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

traverse: To iterate through all the elements of a set performing a similar operation on each.

counter: A variable used to count something, usually initialized to zero and then incremented.

increment: Increase the value of a variable by one. The increment operator in Java is `++`.

decrement: Decrease the value of a variable by one. The decrement operator in Java is `--`.

exception: A run time error. Exceptions cause the execution of a program to terminate.

Appendix A

Contracts/APIs

Every class is characterized by a **contract** that specifies the behavior of its methods – that is, how instances of the class respond to messages. For each method, the contract specifies:

- The name of the method;
- The number, types, and names of the parameters;
- The type of the value returned by the method (or void, if no value is returned);
- The visibility of the method (public, private, protected, package);
- An English description of how an instance of the class behaves when the method is invoked.

The contract defines an **abstraction barrier** between the users and the implementors of the class. A user of the class can expect that objects will behave as described in the contract, but cannot expect anything more than what is specified in the contract. An implementor of the class must ensure that objects fulfill the contract, but need not implement anything more than what is specified in the contract.

In the programming community, another term for a contract is an **Application Programming Interface (API)**. An API is the documentation that allows programmers to use the components of a software library as black-box abstractions. Every class in Java is a library component that is described by an API. The Java API can be daunting for the uninitiated.

The following section include the contracts for some of the Java classes we will work with in this class, as well as for the microworlds developed at Wellesley College.

A.1 Buggle Contract

Constructor

`Buggle ()`

Returns a new buggle at position (1,1) whose heading is Direction.EAST, whose color is Color.red, and whose brush is down.

Instance Methods

`public void forward ()`

Moves this buggle forward one step (in the direction of its current heading). Complains if the buggle is facing a wall. If the Buggle's brush is down, a colored trail appears in its wake.

`public void forward (int n)`

Moves this buggle forward n steps. If the buggle encounters a wall along the way, it will stop and complain. If the Buggle's brush is down, a colored trail appears in its wake.

`public void backward ()`

Moves this buggle backward one step (in the direction opposite to its current heading). Complains if the buggle is facing a wall. If the Buggle's brush is down, a colored trail appears in its wake.

`public void backward (int n)`

Moves this backward n steps. If the buggle encounters a wall along the way, it will stop and complain. If the Buggle's brush is down, a colored trail appears in its wake.

`public boolean isFacingWall ()`

Returns true if this buggle is next to a wall of Buggle world and facing it, false otherwise.

`public void left ()`

Turns this buggle left by 90 degrees.

`public void right ()`

Turns this buggle right by 90 degrees.

`public void brushDown ()`

Lowers this buggle's brush. When the brush is lowered, the buggle leaves a trail when it moves.

`public void brushUp ()`

Raises this buggle's brush. When the brush is raised, the buggle leaves no trail when it moves.

`public boolean isBrushDown ()`

Return true if this Buggle will leave a trail when it moves, false otherwise.

`public void dropBagel ()`

Drops a bagel in the cell currently occupied by this buggle. If there is already a bagel in the cell, tbe buggle complains.

`public void pickUpBagel ()`

Picks up the bagel in the cell currently occupied by this buggle. If there is no bagel in the cell, the buggle complains.

`public boolean isOverBagel ()`

Returns true if there is a bagel in the cell currently occupied by this buggle, false otherwise.

`public Point getPosition ()`

Returns a point that indicates the current position of this buggle in the grid.

`public void setPosition (Point p)`

Changes the position of this buggle to be the point p. Complains if p is not in the grid. Does not leave a trail.

`public Direction getHeading ()`

Returns the heading of this buggle.

`public void setHeading (Direction d)`

Changes the heading of this buggle to be the direction d.

`public Color getColor ()`

Returns the color of this buggle.

`public void setColor (Color c)`

Changes the color of this buggle to be the color c.

`public String toString ()`

Returns a string representation of this buggle.

A.2 BuggleWorld Contract

Instance Method

`public void run ()`

Execute specified buggle actions in this buggle world.

A.3 Point Contract

Constructor

`public Point (int x, int y)`

Returns a point with x-coordinate x and y-coordinate y.

Instance Methods

`public boolean equals (Point p)`

Returns true if this point has the same coordinates as p, and false otherwise.

`public String toString ()`

Returns a string representation of this point.

A.4 Direction Contract

Constants

The following constant directions specify the four compass points:

`public static final Direction NORTH`

`public static final Direction EAST`

`public static final Direction SOUTH`

`public static final Direction WEST`

Instance Methods

`public Direction left ()`

Returns the compass point to the left of this direction.

`public Direction right ()`

Returns the compass point to the right of this direction.

`public Direction opposite ()`

Returns the compass point opposite to this direction.

`public boolean equals (Direction d)`

Returns true if this direction equals d, and false otherwise.

`public String toString ()`

Returns a string representation of this direction.

A.5 PictureWorld Contract

Class methods

`public Picture empty()`

Returns the empty picture.

Methods that rotate Pictures

`public Picture clockwise90 (Picture p)`

Returns a Picture which is the original Picture `p` rotated 90 degrees in the clockwise direction.

`public Picture clockwise180 (Picture p)`

Returns a Picture which is the original Picture `p` rotated 180 degrees in the clockwise direction.

`public Picture clockwise270 (Picture p)`

Returns a Picture which is the original Picture `p` rotated 270 degrees in the clockwise direction.

Methods that flip Pictures around an axes

`public Picture flipHorizontally (Picture p)`

Returns a Picture which is the original Picture `p` flipped around its middle horizontal axis (from (0.0,0.5) to (1.0,0.5)).

`public Picture flipVertically (Picture p)`

Returns a Picture which is the original Picture `p` flipped around its middle vertical axis (from (0.5,0.0) to (0.5,1.0)).

`public Picture flipDiagonally (Picture p)`

Returns a Picture which is the original Picture `p` flipped around its diagonal axis (from (0.0,0.0) to (1.0,1.0)).

`public Picture overlay (Picture p1, Picture p2)`

Returns a Picture which is the result of placing Picture `p1` on top of Picture `p2`.

Methods that place Pictures next to each other

`public Picture beside (Picture p1, Picture p2)`

Returns a Picture which is the result of placing Picture `p1` to the left of Picture `p2` where each Picture takes up half of the screen.

`public Picture beside (Picture p1, Picture p2, double fraction)`

Returns a Picture which is the result of placing Picture `p1` to the left of Picture `p2` where `p1` takes up the specified `fraction` of the screen. `fraction` is a decimal number between 0.0 (none of the screen) and 1.0 (the entire screen).

`public Picture above (Picture p1, Picture p2)`

Returns a Picture which is the result of placing Picture `p1` above Picture `p2` where each Picture takes up half of the screen.

`public Picture above (Picture p1, Picture p2, double fraction)`

Returns a Picture which is the result of placing Picture `p1` above Picture `p2` where `p1` takes up the specified `fraction` of the screen. `fraction` is a decimal number between 0.0 (none of the screen) and 1.0 (the entire screen).

Picture Choice Manipulators

`public void initializePictureChoices ()`

This method is invoked when a PictureWorld applet is created.

`public void addPictureChoice (String name, Picture pic)`

This method adds the choice `name` to the list of Picture choices and associates it with the Picture `pic`.

A.6 Turtle World Contract

Methods with integer parameters

`public void fd (int n)`

Move the turtle forward `n` steps.

`public void bd (int n)`

Move the turtle backward `n` steps.

`public void lt (int angle)`

Turn the turtle to the left `angle` degrees.

`public void rt (int angle)`

Turn the turtle to the right `angle` degrees.

`public void pu`

Raise the turtle's pen.

`public void pd`

Lower the turtle's pen.

Methods with floating-point parameters

There are versions of `fd, bd, lt` and `rt` that take `double` parameters, so you can invoke these methods with either integer or floating-point values.

A.7 IntList Contract

The IntList class describes immutable linked lists of integers. An integer list is either the *empty list* or a (non-empty) *list node* with a *head* component that is an integer and a *tail* component that is another integer list. The lists are *immutable* in the sense that the head or tail of a list node cannot be changed after the list node has been created (via the prepend() method).

There are no public constructor or instance methods for integer lists; they are created and manipulated by the following class methods:

Class Methods

`public static IntList empty()`

Returns an empty integer list.

`public static boolean isEmpty(IntList list)`

Returns **true** if `list` is an empty integer list and **false** if `list` is an integer list node.

`public static IntList prepend (int n, IntList tail)`

Returns a new integer list node whose head is n and whose tail is `tail`.

`public static int head (IntList list)`

Returns the integer that is the head component of the integer list node `list`. Signals an exception if `list` is empty.

`public static IntList tail (IntList list)`

Returns the integer list that is the tail component of the integer list node `list`. Signals an exception if `list` is empty.

A.8 ObjectList Contract

The ObjectList class describes immutable linked lists of objects. An object list is either the empty list or a list node with a head component that is an Object and a tail component that is an ObjectList. ObjectLists are immutable in the sense that the head or tail of a list node cannot be changed after the list node has been created (via the `prepend` method).

There are no public constructor or instance methods for integer lists; they are created and manipulated by the following class methods:

Class Methods

```
public static ObjectList empty()
```

Returns an empty object list.

```
public static boolean isEmpty (ObjectList list)
```

Returns `true` if `list` is an empty list and `false` if list is an object list node.

```
public static ObjectList prepend (Object x, ObjectList list)
```

Returns a new ObjectList whose head is `x` and whose tail is `list`.

```
public static Object head (ObjectList list)
```

Returns the Object that is the head of `list`. Signals an exception if `list` is empty.

```
public static ObjectList tail (ObjectList list)
```

Returns the ObjectList that is the tail of `list`. Signals an exception if `list` is empty.

Appendix B

Debugging

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

- Compile-time errors are produced by the compiler and usually indicate that there is something wrong with the syntax of the program. Example: omitting the semi-colon at the end of a statement.
- Run-time errors are produced by the run-time system if something goes wrong while the program is running. Most run-time errors are Exceptions. Example: an infinite recursion eventually causes a StackOverflowException.
- Semantic errors are problems with a program that compiles and runs, but doesn't do the right thing. Example: an expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, there are some techniques that are applicable in more than one situation.

B.1 Compile-time errors

The compiler is spewing error messages.

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it fails, and it reports spurious errors.

In general, only the first error message is reliable. I suggest that you only fix one error at a time, and then recompile the program. You may find that one semi-colon "fixes" 100 errors. Of course, if you see several legitimate error messages, you might as well fix more than one bug per compilation attempt.

I'm getting a weird compiler message and it won't go away.

First of all, read the error message carefully. It is written in terse jargon, but often there is a kernel of information there that is carefully hidden.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally the error will be prior to the location of the error message, but there are cases where it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Now is a good time to go through the whole program and make sure it is indented properly. I won't say that good indentation makes it easy to find syntax errors, but bad indentation sure makes it harder.

Now, start examining the code for the common syntax errors.

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.

2. Remember that upper case letters are not the same as lower case letters.

3. Check for semi-colons at the end of statements (and no semi-colons after squiggly-braces).

4. Make sure that any strings in the code have matching quotation marks (and that you use double-quotes, not single).

5. For each assignment statement, make sure that the type on the left is the same as the type on the right.

6. For each method invocation, make sure that the arguments you provide are in the right order, and have right type, and that the object you are invoking the method on is the right type.

7. If you are invoking a fruitful method, make sure you are doing something with the result. If you are invoking a void method, make sure you are not *trying* to do something with the result.

8. If you are invoking an object method, make sure you are invoking it on an object with the right type. If you are invoking a class method from outside the class where it is defined, make sure you specify the class name.

9. Inside an object method you can refer to the instance variables without specifying an object. If you try that in a class method, you will get a confusing message like, "Static reference to non-static variable."

If nothing works, move on to the next section...

I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your development environment to make sure the program you are editing is the program the compiler is compiling. If you are not sure, try putting an obvious and deliberate syntax error right at the beginning of the program. Now compile again. If the compiler doesn't find the new error, there is probably something wrong with the way you set up the project.

Otherwise, if you have examined the code thoroughly, it is time for desperate measures. You should start over with a program that you can compile and then gradually add your code back.

- Make a copy of the file you are working on. If you are working on `Fred.java`, make a copy called `Fred.java.old`.
- Delete about half the code from `Fred.java`. Try compiling again.
  - If the program compiles now, then you know the error is in the other half. Bring back about half of the code you deleted and repeat.
  - If the program still doesn't compile, the error must be in this half. Delete about half of the code and repeat.
- Once you have found and fixed the error, start bringing back the code you deleted, a little bit at a time.

This process is called "debugging by bisection." As an alternative, you can comment out chunks of code instead of deleting them. For really sticky syntax problems, though, I think deleting is more reliable—you don't have to worry about the syntax of the comments, and by making the program smaller you make it more readable.

B.2 Run-time errors

My program hangs.

If a program stops and seems to be doing nothing, we say it is "hanging." Often that means that it is caught in an infinite loop or an infinite recursion.

- If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the section titled "Infinite loop."

- Most of the time an infinite recursion will cause the program to run for a while and then produce a StackOverflowException. If that happens, go to the section titled "Infinite recursion."

  If you are not getting a StackOverflowException, but you suspect there is a problem with a recursive method, you can still use the techniques in the infinite recursion section.

- If neither of those things works, start testing other loops and other recursive methods.

- If none of those things works, then it is possible that you don't understand the flow of execution in your program. Go to the section titled "Flow of execution."

Infinite loop

If you think you have an infinite loop and think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition, and the value of the condition.

For example,

```
while (x > 0 && y < 0) {
    // do something to x
    // do something to y

    System.out.println ("x: " + x);
    System.out.println ("y: " + y);
    System.out.println ("condition: " + (x > 0 && y < 0));
}
```

Now when you run the program you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loops keeps going, you will be able to see the values of `x` and `y` and you might figure out why they are not being updated correctly.

Infinite recursion

Most of the time an infinite recursion will cause the program to run for a while and then produce a StackOverflowException.

If you suspect that method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some

condition that will cause the method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case, but the program doesn't seem to be reaching it, add a print statement at the beginning of the method that prints the parameters. Now when you run the program you will see a few lines of output every time the method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each method with a message like "entering method foo," where `foo` is the name of the method.

Now when you run the program it will print a trace of each method as it is invoked.

It is often useful to print the parameters each method receives when it is invoked. When you run the program, check whether the parameters are reasonable, and check for one of the classic errors—providing parameters in the wrong order.

When I run the program I get an Exception.

If something goes wrong during run time, the Java run-time system prints a message that includes the name of the exception, the line of the program where the problem occurred, and a stack trace.

The stack trace includes the method that is currently running, and then the method that invoked it, and then the method that invoked *that*, and so on. In other words, it traces the path of method invocations that got you to where you are.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened.

NullPointerException: You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out what variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an object type, it is initially `null`, until you assign a value to it. For example, this code causes a NullPointerException:

```
Point blank;
System.out.println (blank.x);
```

ArrayIndexOutOfBoundsException: The index you are using to access an array is either negative or greater than `array.length-1`. If you can find the site where the problem is, add a print statement immediately before

it to print the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backwards through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing.

If either one is a parameter, go to the place where the method is invoked and see where the values are coming from.

StackOverFlowException: See "Infinite recursion."

I added so many print statements I get inundated with output.

One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: either simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the error.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the error is in a deeply-nested part of the program, try rewriting that part with simpler structure. If you suspect a large method, try splitting it into smaller methods and test them separately.

Often the process of finding the minimal test case leads you to the bug. For example, if you find that a program works when the array has an even number of elements, but not when it has an odd number, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

B.3 Semantic errors

My program doesn't work.

In some ways semantic errors are the hardest, because the compiler and the run-time system provide no information about what is wrong. Only you know what the program was supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is

actually doing. One of the things that makes this hard is that computers run so fast. You will often wish that you could slow the program down to human speed, but there is no straightforward way to do that, and even if there were, it is not really a good way to debug.

Here are some questions to ask yourself:

- Is there something the program was supposed to do, but doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should. Add a print statement to the beginning of the suspect methods.

- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.

- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to built-in Java methods. Read the documentation for the methods you invoke. Try out the methods by invoking the methods directly with simple test cases, and check the results.

In order to program, you need to have a mental model of how programs work. If your program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the classes and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

Here are some common semantic errors that you might want to check for:

- If you use the assignment operator, `=`, instead of the equality operator, `==`, in the condition of an `if`, `while` or `for` statement, you might get an expression that is sytactically legal, but it doesn't do what you expect.

- When you apply the equality operator, `==`, to an object, it checks shallow equality. If you meant to check deep equality, you should use the `equals` method (or define one, for user-defined objects).

- Some Java libraries expect user-defined objects to define methods like `equals`. If you don't define them yourself, you will inherit the default behavior from the parent class, which may not be what you want.

- In general, inheritance can cause subtle semantic errors, because you may be executing inherited code without realizing it. Again, make sure you understand the flow of execution in your program.

I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
bobby.setColor (bobby.getColor().darker());
```

Can be rewritten as

```
Color bobbyColor = bobby.getColor();
Color darker = bobbyColor.darker ();
bobby.setColor (darker);
```

The explicit version is easier to read, because the variable names provide additional documentation, and easier to debug, because we can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Java, you might write

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit.

```
double y = x / (2 * Math.PI);
```

Any time you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intend); it will also be more readable for other people who haven't memorized the rules of precedence.

I've got a method that doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of

```
public Picture fourPics (Picture p1, Picture p2, Picture p3, Picture p4){
    return above (beside (p1, p2), beside (p3, p4));
}
```

You could write

```
public Picture fourPics (Picture p1, Picture p2, Picture p3, Picture p4){
    Picture row1 = beside (p1, p2);
    Picture row2 = beside (p3, p4);
    Picture grid = above (row1, row2);
    return grid;
}
```

Now you have the opportunity to print (or display) any of `row1`, `row2` or `grid` before returning.

My print statement isn't doing anything

If your use the `println` method, the output gets displayed immediately, but if you use `print` (at least in some environments) the output gets stored without being displayed until the next newline character gets output. If the program terminates without producing a newline, you may never see the stored output.

If you suspect that this is happening to you, try changing all the `print` statements to `println`.

I'm really, really stuck and I need help

First of all, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and/or rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backwards").
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and I let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. You program should be as simple as possible, and you

should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need.

- What kind of bug is it? Compile-time, run-time, or semantic?
- If the bug occurs at compile-time or run-time, what is the error message, and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, in this class the goal is not to make the program work. The goal is to learn how to make the program work.

Appendix C

Program development plan

If you are spending a lot of time debugging, it is probably because you do not have an effective **program development plan**.

A typical, bad program development plan goes something like this:

1. Write an entire method.
2. Write several more methods.
3. Try to compile the program.
4. Spend an hour finding syntax errors.
5. Spend an hour finding run time errors.
6. Spend three hours finding semantic errors.

The problem, of course, is the first two steps. If you write more than one method, or even an entire method, before you start the debugging process, you are likely to write more code than you can debug.

If you find yourself in this situation, the *only* solution is to remove code until you have a working program again, and then gradually build the program back up. Beginning programmers are often unwilling to do this, because their carefully crafted code is precious to them. To debug effectively, you have to be ruthless!

Here is a better program development plan:

1. Start with a working program that does something visible, like printing something.
2. Add a small number of lines of code at a time, and test the program after every change.
3. Repeat until the program does what it is supposed to do.

After every change, the program should produce some visible effect that demonstrates the new code. This approach to programming can save a lot of time. Because you only add a few lines of code at a time, it is easy to find syntax errors. Also, because each version of the program produces a visible result, you are constantly testing your mental model of how the program works. If your mental model is erroneous, you will be confronted with the conflict (and have a chance to correct it) before you have written a lot of erroneous code.

One problem with this approach is that it is often difficult to figure out a path from the starting place to a complete and correct program.

I will demonstrate by developing a method called `isIn` that takes a String and a Vector, and that returns a boolean: `true` if the String appears in the list and `false` otherwise.

1. The first step is to write the shortest possible method that will compile, run, and do something visible:

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn");
    return false;
}
```

   Of course, to test the method we have to invoke it. In `main`, or somewhere else in a working program, we need to create a simple test case.

   We'll start with a case where the String appears in the vector (so we expect the result to be `true`).

```
public static void main (String[] args) {
    Vector v = new Vector ();
    v.add ("banana");
    boolean test = isIn ("banana", v);
    System.out.println (test);
}
```

   If everything goes according to plan, this code will compile, run, and print the word `isIn` and the value `false`. Of course, the answer isn't correct, but at this point we know that the method is getting invoked and returning a value.

   In my programming career, I have wasted way too much time debugging a method, only to discover that it was never getting invoked. If I had used this development plan, it never would have happened.

2. The next step is to check the parameters the method receives.

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
```

```
    System.out.println ("in the vector " + v);
    return false;
}
```

The first print statement allows us to confirm that `isIn` is looking for the right word. The second statement prints a list of the elements in the vector.

To make things more interesting, we might add a few more elements to the vector:

```
public static void main (String[] args) {
    Vector v = new Vector ();
    v.add ("apple");
    v.add ("banana");
    v.add ("grapefruit");
    boolean test = isIn ("banana", v);
    System.out.println (test);
}
```

Now the output looks like this:

```
isIn looking for banana
in the vector [apple, banana, grapefruit]
```

Printing the parameters might seem silly, since we know what they are supposed to be. The point is to confirm that they are what we think they are.

3. To traverse the vector, we can take advantage of the code from Section 9.8. In general, it is a great idea to reuse code fragments rather than writing them from scratch.

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);

    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
    }
    return false;
}
```

Now when we run the program it prints the elements of the vector one at a time. If all goes well, we can confirm that the loop examines all the elements of the vector.

4. So far we haven't given much thought to what this method is going to do. At this point we probably need to figure out an algorithm. The simplest algorithm is a linear search, which traverses the vector and compares each element to the target word.

   Happily, we have already written the code that traverses the vector. As usual, we'll proceed by adding just a few lines at a time:

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);

    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            System.out.println ("found it");
        }
    }
    return false;
}
```

   As always, we use the `equals` method to compare Strings, not the `==` operator!

   Again, I added a print statement so that when the new code executes it produces a visible effect.

5. At this point we are pretty close to working code. The next change is to return from the method if we find what we are looking for:

```
public static boolean isIn (String word, Vector v) {
    System.out.println ("isIn looking for " + word);
    System.out.println ("in the vector " + v);

    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            System.out.println ("found it");
            return true;
        }
    }
    return false;
}
```

   If we find the target word, we return `true`. If we get all the way through the loop without finding it, then the correct return value is `false`.

If we run the program at this point, we should get

```
isIn looking for banana
in the vector [apple, banana, grapefruit]
apple
banana
found it
true
```

6. The next step is to make sure that the other test cases work correctly. First, we should confirm that the method returns `false` if the word in not in the vector.

   Then we should check some of the typical troublemakers, like an empty vector (one with size 0) and a vector with a single element. Also, we might try giving the method an empty String.

   As always, this kind of testing can help find bugs if there are any, but it can't tell you if the method is correct.

7. The penultimate step is to remove or comment out the print statements.

```
public static boolean isIn (String word, Vector v) {
    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            return true;
        }
    }
    return false;
}
```

   Commenting out the print statements is a good idea if you think you might have to revisit this method later. But if this is the final version of the method, and you are convinced that it is correct, you should remove them.

   Removing the comments allows you to see the code most clearly, which can help you spot any remaining problems.

   If there is anything about the code that is not obvious, you should add comments to explain it. Resist the temptation to translate the code line by line. For example, no one needs this:

```
// if word equals s, return true
if (word.equals (s)) {
    return true;
}
```

You should use comments to explain non-obvious code, to warn about conditions that could cause errors, and to document any assumptions that are built into the code. Also, before each method, it is a good idea to write an abstract description of what the method does.

8. The final step is to examine the code and see if you can convince yourself that it is correct.

   At this point we know that the method is syntactically correct, because it compiles.

   To check for run time errors, you should find every statement that can cause an error and figure out what conditions cause the error.

   The statements in this method that can produce a run time error are:

`v.size()`	if `v` is `null`.
`word.equals (s)`	if `word` is `null`.
`(String) v.get(i)`	if `v` is `null` or `i` is out of bounds, or the ith element of `v` is not a String.

   Since we get `v` and `word` as parameters, there is no way to avoid the first two conditions. The best we can do is check for them.

```
public static boolean isIn (String word, Vector v) {
    if (v == null || word == null) return false;
    for (int i=0; i<v.size(); i++) {
        System.out.println (v.get(i));
        String s = (String) v.get(i);
        if (word.equals (s)) {
            return true;
        }
    }
    return false;
}
```

   In general, it is a good idea for methods to make sure their parameters are legal.

   The structure of the `for` loop ensures that `i` is always between 0 and `v.size()-1`. But there is no way to ensure that the elements of `v` are Strings. On the other hand, we can check them as we go along. The `instanceof` operator checks whether an object belongs to a class.

```
    Object obj = v.get(i);
    if (obj instanceof String) {
        String s = (String) v.get(i);
    }
```

This code gets an object from the vector and checks whether it is a String. If it is, it performs the typecast and assigns the String to `s`.

As an exercise, modify `isIn` so that if it finds an element in the vector that is not a String, it skips to the next element.

If we handle all the problem conditions, we can prove that this method will not cause a run time error.

We haven't proven yet that the method is semantically correct, but by proceeding incrementally, we have avoided many possible errors. For example, we already know that the method is getting parameters correctly and that the loop traverses the entire vector. We also know that it is comparing Strings successfully, and returning `true` if it finds the target word. Finally, we know that if the loop exists, the target word cannot be in the vector.

Short of a formal proof, that is probably the best we can do.

Index