Think Java @ Emory

How to Think Like a Computer Scientist

Valerie Henderson Summet

5.1.2

Attributions

This book is based on Allen B. Downey's book, *Think Java: How to Think Like a Computer Scientist*, version 5.1.2, 2012. It has been heavily modified by the current authors.

Many of the modifications to this work were based on a series of lecture notes by my colleague Dr. SY Cheung who did much work on Emory's Computer Science 170 course. I have borrowed heavily from his notes and examples as well.

Moreover, the book was constructed using Allen Downey's LaTeXtemplates and source code (available at http://thinkapjava.com. Again, the source have been heavily modified.

Think Java: How to Think Like a Computer Scientist

Valerie Henderson Summet

December 2, 2015

Contents

Listings

Chapter 1

The way of the program

The goal of this book 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 Python, C or C++, and Perl.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes called machine language or assembly language. Loosely-speaking, computers can only run 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 time, which is a small disadvantage of high-level languages.

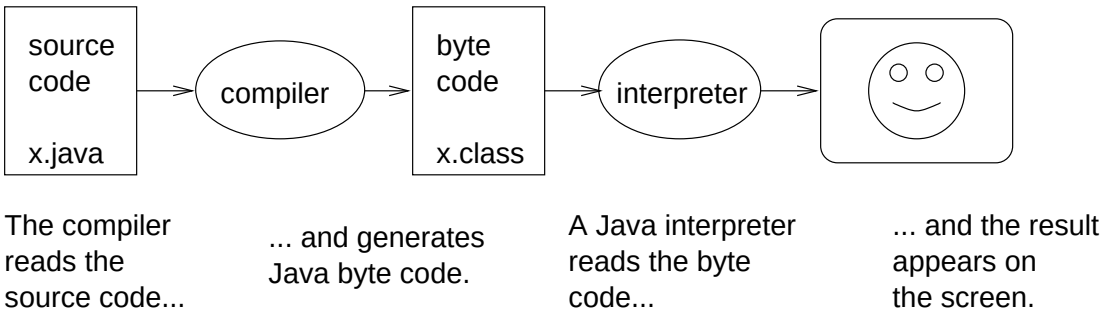
The advantages are enormous. First, it is *much* easier to program in a high-level language: the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers 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** and **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.

A compiler is a program that reads a high-level program and translates it all at once, before running any of the commands. Often you compile the program as a separate step, and then run 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**.

Java is both compiled and interpreted. Instead of translating programs into machine language, the Java compiler generates **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 program on one machine, transfer the byte code to another machine, and then interpret the byte code on the other machine. This ability is an advantage of Java over many other high-level languages.



Although this process may seem complicated, in most program development environments these steps are automated for you. Usually you will only have to write a program and press a button or type a single command to compile and run it. On the other hand, it is useful to know what steps are happening in the background, so 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[1]. 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, which we will call **statements**, look different in different programming languages, but there are a few basic operations most languages perform:

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 run the appropriate sequence of statements.

[1]This definition does not apply to all programming languages; for alternatives, see http://en.wikipedia.org/wiki/Declarative_programming.

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

That's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of statements that perform these operations. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic operations.

1.3 What is debugging?

For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

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

1.3.1 Syntax 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 weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

1.3.2 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.

Java tends to be a **safe** language, which means that the compiler catches a lot of errors. So run-time errors are rare, especially for simple programs.

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.

1.3.3 Logic errors and semantics

The third type of error is the **logic** or **semantic** error. If there is a logic error in your program, it will compile and run without generating 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 semantics, or meaning of the program, are wrong. Identifying logic errors can be tricky because you have to work backwards, looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

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

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 make more suggestions about debugging and other programming practices.

1.4 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 order on them); they evolved naturally.

Formal languages are languages 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.**

Formal languages have strict rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but 3\$ = 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\$ = 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 rule pertains to the structure of a statement; that is, the way the tokens are arranged. The statement 3\$ = is structurally

illegal, because you can't have an equals sign at the end of an equation. 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**.

Although formal and natural languages have features in common—tokens, structure, syntax and semantics—there are 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 unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: To make up for ambiguity and reduce misunderstandings, natural languages are often redundant. Formal languages are more concise.

literalness: Natural languages are full of idiom and metaphor. 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 common and deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning.

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 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.

1.5 The first program

Traditionally the first program people write in a new language is called "hello world" because all it does is display the words "Hello, World." In Java, this program looks like:

```
public class Hello {
  // main: generate some simple output
  public static void main(String[] args) {
    System.out.println("Hello, world.");
  }
}
```

Program 1.1: Hello.java

This program includes features that are hard to explain to beginners, but it provides a preview of topics we will see in detail later.

Java programs are made up of **class definitions**, which have the form:

```
public class CLASSNAME {

  public static void main (String[] args) {
    STATEMENTS
  }
}
```

Here `CLASSNAME` indicates a name chosen by the programmer. The class name in the example is `Hello`.

`main` is a **method**, which is a named collection of statements. The name `main` is special; it marks the place in the program where execution begins. When the program runs, it starts at the first statement in `main` and ends when it finishes the last statement.

`main` can have any number of statements, but the example has one. It is a **print statement**, meaning that it displays a message on the screen. Confusingly, "print" can mean "display something on the screen," or "send something to the printer." In this book I won't say much about sending things to the printer; we'll do all our printing on the screen. The print

statement ends with a semi-colon (;).

When considering all of these new parts, it may be useful to keep the analogy of a book in your head. A *book* is composed of *chapters*. *Chapters* are composed of *paragraphs*. *Paragraphs* are composed of *sentences*. We will utilize a similar structure in Java programming. *Programs* are composed of *classes*. *Classes* are composed of *methods*. And *methods* are composed of *statements*. It's worth noting that most programs we write will be composed of only one class, but later in the course, we will begin to write multiple-class programs.

`System.out.println` is a method provided by one of Java's libraries. A **library** is a collection of class and method definitions.

Java uses curly-braces ({ and }) to group things together. The outermost set of curly-braces (lines 1 and 7) contain the class definition, and the inner braces (lines 4 and 6) contain the definition of `main`.

Line 3 begins with `//`. That means it's a **comment**, which is a bit of English text that you can put a program, usually to explain what it does. When the compiler sees `//`, it ignores everything from there until the end of the line.

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 run. 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 run 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 run.

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.

statement: A part of a program that specifies a computation.

print statement: A statement that causes output to be displayed on the screen.

comment: A part of a program that contains information about the program, but that has no effect when the program runs.

method: A named collection of statements.

library: A collection of class and method definitions.

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.

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.

logic 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.

1.7 Exercises

Exercise 1.1. Computer scientists have the annoying habit of using common English words to mean something other than their common English meaning. For example, in English, statements and comments are the same thing, but in programs they are different.

The glossary at the end of each chapter is intended to highlight words and phrases that have special meanings in computer science. When you see familiar words, don't assume that you know what they mean!

1. In computer jargon, what's the difference between a statement and a comment?

2. What does it mean to say that a program is portable?

3. What is an executable?

Exercise 1.2. Before you do anything else, find out how to compile and run a Java program in your environment. Some environments provide sample programs similar to the example in Section 1.5.

1. Type in the "Hello, world" program, then compile and run it.

2. Add a print statement that prints a second message after the "Hello, world!". Something witty like, "How are you?" Compile and run the program again.

3. Add a comment to the program (anywhere), recompile, and run it again. The new comment should not affect the result.

This exercise may seem trivial, but it is the starting place for many of the programs we will work with. To debug with confidence, you have to have confidence in your programming environment. In some environments, it is

easy to lose track of which program is executing, and you might find yourself trying to debug one program while you are accidentally running another. Adding (and changing) print statements is a simple way to be sure that the program you are looking at is the program you are running.

Exercise 1.3. It is a good idea to commit as many errors as you can think of, so that you see what error messages the compiler produces. Sometimes the compiler tells you exactly what is wrong, and all you have to do is fix it. But sometimes the error messages are misleading. You will develop a sense for when you can trust the compiler and when you have to figure things out yourself.

1. Remove one of the open curly-braces.
2. Remove one of the close curly-braces.
3. Instead of `main`, write `mian`.
4. Remove the word `static`.
5. Remove the word `public`.
6. Remove the word `System`.
7. Replace `println` with `Println`.
8. Replace `println` with `print`. This one is tricky because it is a logic error, not a syntax error. The statement `System.out.print` is legal, but it may or may not do what you expect.
9. Delete one of the parentheses. Add an extra one.

Chapter 2

Variables and types

2.1 More printing

You can put as many statements as you want in `main`; for example, to print more than one line:

```
public class MoreHello {
  // Generates some simple output.

  public static void main(String[] args) {
    System.out.println("Hello, world.");     // print one line
    System.out.println("How are you?");      // print another
  }
}
```

Program 2.1: MoreHello.java

As this example demonstrates, you can put comments at the end of a line, as well as on a line by themselves.

The phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of characters. 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 moves the cursor to the next line of the display. As an analogy, think of typing on an old-time typewriter and reaching the edge of the paper; you had to hit the "Return" key to start a new line. The behavior here is largely the same. The next time `println` is invoked, the new text appears on the next line.

To display the output from multiple print statements all on one line, use `print`:

```
public class PrintHello {

   // Generates some simple output
   // all on the same line.
   public static void main(String[] args) {
      System.out.print("Goodbye, ");
      System.out.println("cruel world!");
   }
}
```

Program 2.2: PrintHello.java

The output appears on a single line as `Goodbye, cruel world!`. 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.

Spaces that appear outside of quotation marks generally do not affect the behavior of the program. For example, I could have written:

```
public class UglyHello {
public static void main(String[] args) {
System.out.print("Goodbye, "      );
System.out.println("cruel world!");
}
}
```

Program 2.3: UglyHello.java

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program's behavior either, so I could have written:

```
public class ReallyUglyHello { public static void main(String[] args) {
System.out.print("Goodbye, "       ); System.out.println
("cruel world!");}}
```

Program 2.4: ReallyUglyHello.java

That would work, too, but the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate errors.

These example show that Java is a **format-free language**. This means that the positioning of the characters in the program is insignificant and does not effect the compilation or execution of the programs. Other languages

(notably Python) are formatted languages, which means that the spacing of the program is important! In formatted languages, the structure and spacing convey meaning to the interpretter or compiler.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a named location in the computer's memory that stores a **value**. Values are things that can be printed, stored and (as we'll see later) operated on. The strings we have been printing (`"Hello, World."`, `"Goodbye, "`, etc.) are values.

To store a value, you have to create a variable. Since the values we want to store are strings, we declare that the new variable is a string:

```
1    String bob;
```

This statement is a **declaration**, because it declares that the variable named `bob` has the type `String`. Each variable has a type that determines what kind of values it can store. For example, the `int` type can store integers, and the `String` type can store strings.

Some types begin with a capital letter and some with lower-case. We will learn the significance of this distinction later, but for now you should take care to get it right. There is no such type as `Int` or `string`, and the compiler will object if you try to make one up.

To create an integer variable, the syntax is `int bob;`, where `bob` is the arbitrary name you made up for the variable. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
1    String firstName;
2    String lastName;
3    int hour, minute;
```

you could guess what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `second` are both integers (`int` type).

Java has some rules about what you can name your variables. If you violate these rules, your work will not compile.

Rules of naming variables include:

1. Variable names are case sensitive. `Bob` and `bob` are different names.

2. Variable names cannot contain spaces or special symbols except for `$` or `_`.

3. Variable names must start with a letter or `$` or `_`.

4. Subsequent characters can be letters, numbers, `$` or `_`.

5. Variable names cannot be keywords. (More about this in Section 2.6).

The Java programming community has also created some rules about how to name variables. However, these are just conventions. Your code will still compile and run if you violate them, but for a variety of reasons, you usually want to follow conventions.

Conventions to be followed when naming variables include:

1. Do not begin variables names with `$` or `_`. Beginning variable names with these symbols has special meaning to programmers in other languages and is confusing.

2. Variable names should be descriptive to the task. In other words, they should give a clue about what values they will be storing.

3. Variable names should begin with a lowercase letter and subsequent words should be "camel caps" which is a cute name for `jammingWordsTogetherLikeThis`.

2.3 Assignment

Now that we have created variables, we want to store values. We do that with an **assignment statement**.

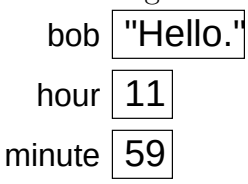
```
1    bob = "Hello.";      // give bob the value "Hello."
2    hour = 11;           // assign the value 11 to hour
3    minute = 59;         // set minute to 59
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.

- When you make an assignment to a variable, you give it a value.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This figure shows the effect of the three assignment statements:



As a general rule, a variable has to have the same type as the value you assign it. You cannot store a `String` in `minute` or an integer in `bob`.

On the other hand, that rule can be confusing, because there are many ways that you can convert values from one type to another, and Java sometimes converts things automatically. For now you should remember the general rule, and we'll talk about exceptions later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, `bob` can contain the string `"123"`, which is made up of the characters `1`, `2` and `3`, but that is not the same thing as the *number* `123`.

```
bob = "123";      // legal
bob = 123;        // not legal
```

2.4 Printing variables

You can print the value of a variable using `println` or `print`:

```
public class HelloVariable {
  public static void main(String[] args) {
    String firstLine;
    firstLine = "Hello, again!";
    System.out.println(firstLine);
  }
}
```

Program 2.5: HelloVariable.java

This program creates a variable named `firstLine`, assigns it the value `"Hello, again!"` and then prints that value. When we talk about "printing a variable," we mean printing the *value* of the variable. To print the *name* of a variable, you have to put it in quotes. For example: `System.out.println("firstLine");`

For example, you can write

```
    String firstLine;
    firstLine = "Hello, again!";
    System.out.print("The value of the variable firstLine is: ");
    System.out.println(firstLine);
```

The output of these statements are:

```
The value of the variable firstLine is: Hello, again!
```

The syntax for printing a variable is the same regardless of the variable's type. For example:

```
    int hour, minute;
    hour = 11;
    minute = 59;
    System.out.print("The current time is ");
    System.out.print(hour);
    System.out.print(":");
    System.out.print(minute);
    System.out.println(".");
```

The output of these statements are:

```
The current time is 11:59.
```

WARNING: To put multiple values on the same line, is common to use several `print` statements followed by a `println`. But you have to remember the `println` at the end. In many environments, the output from `print` is stored without being displayed until `println` is invoked, at which point the entire line is displayed at once. If you omit `println`, the program may terminate without displaying the stored output!

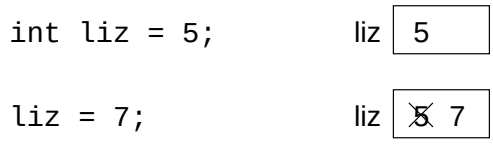
2.5 Multiple assignment

You can have multiple assignment statements for the same variable. The effect is to replace the old value with the new.

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

The output of this program is `57`, because the first time we print `liz` her value is 5, and the second time her 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:



It's important to note that multiple assignment statements replace the old value with a new value. Once you have assigned a new value, there is no way to get the old value back. It has been overwritten in the computer's memory.

Multiple assignment is extremely useful. However, if the values of variables change often, it can make the code difficult to read and debug so be sure to pay attention when it is used. Remember that each statement is evaluated and executed with the current value of each variable, never previous values which have been overwritten.

2.6 Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are reserved in Java because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include `public`, `class`, `void`, `int`, and many more.

The complete list is available at
http://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html.
This site, provided by Oracle, includes Java documentation I refer to throughout the book.

Rather than memorize the list, I suggest you take advantage of a feature provided in many Java development environments: code highlighting. As you type, parts of your program should appear in different colors. For example, in `gedit` keywords appear green, strings and other values pink, comments blue, and other code black. If you type a variable name and it turns green, watch out! You might get some strange behavior from the compiler.

2.7 Operators

Operators are symbols used to instruct the computer to do some kind of computation. In fact, we've already seen an example of an operator! The assignment operator (=) was used to instruct the computer to assign a value to a variable.

Some other operators represent common mathematical computations like addition and multiplication. Many operators in Java do what you expect them to do because they are common mathematical symbols. For example, the operator for addition is `+`. Subtraction is `-`, multiplication is `*`, and division is `/`.

An **expression** is a syntactically valid combination of variables, values, and operators. Variables are replaced with their values before the computation is performed.

The following are all examples of expressions:

```
1+1        hour-1       hour*60 + minute     minute/60
```

We say that we **evaluate** expressions when we determine that the expression `1+1` yields `2` or that `hour-1` yields `10`. In a programming language, evaluating an expression will yield a value. This value will have a type, as we discussed previously. So we evaluate the expression `1+1` and it yields the value `2` which has the type integer (or `int`).

Since expressions give us values, we can print out these values too! When the Java programming language encounters a statement like

```
System.out.println(hour + 2 * 3);
```

Java evaluates the expression inside the parentheses and determines the resulting value. Java then displays that value for us.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. Work through the following code and try to figure out what will be displayed.

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print("Number of minutes since midnight: ");
System.out.println(hour*60 + minute);
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute/60);
```

These statements generate the output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is expected, but the second line is odd. The value of `minute` is 59, and 59 divided by 60 is 0.98333, not 0. The problem is that Java is performing **integer division**.

When both **operands** are integers (operands are the values operators operate on), the result is also an integer, and by convention integer division always rounds *down*, even in cases like this where the next integer is so close.

An alternative is to calculate a percentage rather than a fraction:

```
1    System.out.print("Percentage of the hour that has passed: ");
2    System.out.println(minute*100/60);
```

The result is:

```
Percentage of the hour that has passed: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. To get a more accurate answer, we can use a different type of variable, called floating-point, that can store fractional values. We'll get to that in the next chapter.

2.8 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but the mathematical operators (`*, /, +, -`) follow some of the common rules you have previously learned in your mathematics courses:

- Multiplication and division happen before addition and subtraction. So `2*3-1` yields 5, not 4, and `2/3-1` yields -1, not 1 (remember that in integer division `2/3` is 0).
- If the operators have the same precedence they are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding `5900/60`, which in turn yields `98`. If the operations had gone from right to left, the result would be `59*1` which is `59`, which is wrong.

- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so `2 *(3-1)` is 4. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

2.9 Operators for `Strings`

In general you cannot perform mathematical operations on `String`s, even if the strings look like numbers. The following are illegal (assuming that `bob` has type `String`) and would trigger a compiler syntax error.

```
bob - 1          "Hello"/123      bob * 4
```

By the way, can you tell by looking at those expressions whether `bob` is an integer or a string? Nope. The only way to tell the type of a variable is to look at the place where it is declared.

Interestingly, the `+` operator *does* work with `String`s, but it might not do what you expect. For `String`s, the `+` operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. So the expression:

```
"Hello, " + "world."
```

yields the string `"Hello, world."` and

```
bob + "ism"
```

adds the suffix *ism* to the end of whatever value `bob` has.

2.10 Composition

So far we have looked at the elements of a programming language—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** (combine) them. As we saw earlier, we know how to multiply numbers and we know how to print and we can combine them in a single statement:

```
System.out.println(17 * 3);
```

`51` will be printed to the screen since the expression `17 * 3` evaluates to 51. Any syntactically valid expression involving operators, numbers, strings and variables can be used inside a print statement. We've already seen one example:

```
System.out.println(hour*60 + minute);
```

To know what will be displayed on the screen, we must be able to evaluate the expression `hour*60 + minute`.

But you can also put arbitrary expressions on the right-hand side of an assignment statement:

```
int percentage;
percentage = (minute * 100) / 60;
```

This statement has more operators than you might think. In fact it has four. The operators are `= () * /`. The assignment operator has a lower precedence then the mathematical operators, including the parentheses. Everything on the right hand side of the assignment operator will be evaluated, and a numerical value will be calculated. This value will then be assigned to the variable `percentage`.

This ability may not seem impressive now, but we will see examples where composition allows us to express complex computations neatly and concisely.

WARNING: The left side of an assignment has to be a *variable* name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values. So the following is illegal: `minute+1 = hour;`.

2.11 Glossary

format-free language: A programming language in which the structure of the program is inconsequential to the compilation and execution of the program.

variable: A named storage location for values. All variables have a type, which is declared when the variable is created.

value: A number or string (or other thing to be named later) that can be stored in a variable. Every value belongs to a type.

type: A set of values. The type of a variable determines which values can be stored there. The types we have seen are integers (`int` in Java) and strings (`String` in Java).

keyword: A reserved word used by the compiler to parse programs. You cannot use keywords, like `public`, `class` and `void` as variable names.

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

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

expression: A combination of variables, operators and values that represents a single value. Expressions also have types, as determined by their operators and operands.

evaluate: To determine the result (both type and value) of an expression.

operator: A symbol that represents a computation like addition, multiplication or string concatenation.

operand: One of the values on which an operator operates.

precedence: The order in which operations are evaluated.

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

composition: The ability to combine simple expressions and statements into compound statements and expressions to represent complex computations concisely.

2.12 Exercises

Exercise 2.1. If you are using this book in a class, you might enjoy this exercise: find a partner and play "Stump the Chump":

Start with a program that compiles and runs correctly. One player turns away while the other player adds an error to the program. Then the first player tries to find and fix the error. You get two points if you find the error without compiling the program, one point if you find it using the compiler, and your opponent gets a point if you don't find it.

Exercise 2.2. Take a look at the following variable names which we want to use in a program which will calculate sale prices for a clothing store.

price	$price	$price@sale	price_at_sale
Price	Price15Percent	price15	15PercentPrice
PRICE	_price	15price	p15

1. Which are valid variable names?

2. Which are valid but violate Java programming conventions? Which convention is violated?

Exercise 2.3.

1. Create a new program named `Date.java`. Copy or type in something like the "Hello, World" program and make sure you can compile and run it.

2. Following the example in Section 2.4, write a program that creates variables named `day`, `date`, `month` and `year`. `day` will contain the day of the week and `date` will contain the day of the month. What type is each variable? Assign values to those variables that represent today's date.

3. Print the value of each variable on a line by itself. This is an intermediate step that is useful for checking that everything is working so far.

4. Modify the program so that it prints the date in standard American form: `Saturday, July 16, 2011`.

5. Modify the program again so that the total output is:

```
American format:
Saturday, July 16, 2011
European format:
Saturday 16 July, 2011
```

The point of this exercise is to use string concatenation to display values with different types (`int` and `String`), and to practice developing programs gradually by adding a few statements at a time.

Exercise 2.4. 1. Create a new program called `Time.java`. From now on, I won't remind you to start with a small, working program, but you should.

2. Following the example in Section 2.6, create variables named `hour`, `minute` and `second`, and assign them values that are roughly the current time. Use a 24-hour clock, so that at 2pm the value of `hour` is 14.

3. Make the program calculate and print the number of seconds since midnight.

4. Make the program calculate and print the number of seconds remaining in the day.

5. Make the program calculate and print the percentage of the day that has passed.

6. Change the values of `hour`, `minute` and `second` to reflect the current time (I assume that some time has elapsed), and check to make sure that the program works correctly with different values.

The point of this exercise is to use some of the arithmetic operations, and to start thinking about compound entities like the time of day that that are represented with multiple values. Also, you might run into problems computing percentages with `ints`, which is the motivation for floating point numbers in the next chapter.

HINT: you may want to use additional variables to hold values temporarily during the computation. Variables like this, that are used in a computation but never printed, are sometimes called intermediate or temporary variables.

Chapter 3

Working with types

3.1 Floating-point

In the last chapter we had some problems dealing with numbers that were not integers. We worked around the problem by measuring percentages instead of fractions, but a more general solution is to use floating-point numbers, which can represent fractions as well as integers. In Java, the floating-point type is called `double`, which is short for "double-precision."

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;
```

It is also legal to declare a variable and assign a value to it at the same time:

```
int x = 1;
String empty = "";
double pi = 3.14159;
```

This syntax is common; a combined declaration and assignment is sometimes called an **initialization**.

Although floating-point numbers are useful, they are 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?

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 strictly speaking, you are not allowed to make assignments be-

tween 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 get the value `0.333333`, which is a legal floating-point value, but in fact it gets `0.0`. The reason is that the expression on the right side of the assignment operator involves two integers, so Java does *integer* division, which yields the integer value `0`. When the integer value `0` is then 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.

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.

3.2 Strict typing

We've been talking about the fact that Java is a **strictly typed language**. This means that Java strictly enforces data types and only performs operations when the types are the same. Additionally, all variables must be declared and we, the programmers, must be specific about what type that variable has. After it has been declared, we cannot change the type of the variable. However, we can change the value of the variable using the assignment operator as we saw in the last chapter.

Although we've only learned about `int` and `double` so far, Java actually provides four types for storing non-fractional numbers and two types for storing floating point numbers. Each type uses a different amount of memory in the computer and thus has a different capacity. The following table lists the different types, the amount of computer memory that a variable of a given type would use, and the largest and smallest values you can represent with that type.

Java Type	Amount of memory used	Smallest Value	Largest Value
`byte`	8 bits	-128	127
`short`	16 bits	-32768	32767
`int`	32 bits	-2,147,483,648	2,147,483,647
`long`	64 bits	-9,223,372,036,854,775,808	9 ,223,372,036,854,775,8
`float`	32 bits	*	*
`double`	64 bits	*	*

** Exact values are beyond the scope of this class.*

So now we know that certain types use more space in computer memory and thus are capable of storing larger values. But what does this have to do with typing and whether or not Java will allow us to mix operands of different types?

The rule of thumb is that Java will convert to a *more* precise type for you automatically. This process is called **promotion** and we say that Java promotes the less precise value to the type that is more precise. The opposite process of converting to a *less* precise type is called **demotion**. Java will not automatically demote for you. To make this more concrete, let's look at conversion between a `double` (more precise) and `int` (less precise).

3.3 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. In other words, the values `1` and `1.0` represent the same mathematical quantity and no information is lost by converting the integer value to the floating point value. Any value represented as an integer can be represented as a floating point number simply by adding .0 to it.

On the other hand, going from a `double` to an `int` requires rounding off. For example, we can't convert the floating point value `1.5` to an integer without losing some precision. Java doesn't perform this demotion 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 **casting operator**. **Casting** (also called **typecasting**) is so called because it allows you to take a value of one type and "cast" (convert) it into another type (in the sense of molding or reforming).

The syntax for casting is to put the name of the type in parentheses and use it as an operator. For example,

```
1    double pi = 3.14159;
2    int x = (int) pi;
```

The `(int)` operator on line 2 has the effect of converting what follows into an integer, so the variable `x` gets the value 3. Also, note that we are not specifying the type! Rather, since the type is enclosed in parentheses, we are using a casting operator.

Casting takes precedence over arithmetic operations, so in the following example, the value of `pi` gets converted to an integer first, and the result is 60.0, not 62.

```
1    double pi = 3.14159;
2    double x = (int) pi * 20.0;
```

Converting to an integer always rounds down, even if the fraction part is 0.99999999. These behaviors (precedence and rounding) can make casting error-prone.

3.4 Promotion and demotion in expressions

Now let's consider a slightly more complex example:

```
1  double x = 8 + 19 + 4.5;
2  double y = (int)4.3 + (int)5.8;
3  double z = (int)4.3 + 5.8;
```

To calculate the values assigned to the variables `x`, `y`, and `z`, we must pay close attention to the operators. And these assignment statements are loaded with operators! Line 2 has 4 operators in it and all these operators are executed at different times. We need to know that the assignment operator has the lowest priority and will execute last, after all the operators on the right-hand side of the `=` have executed. While evaluating the expression

on the right-hand side of the assignment operator, we must know that the casting operator has a higher priority than the arithmetic operators.

To help us understand the process, I've rewritten the code after the casting operators have executed:

```
1  double x = 8 + 19 + 4.5;
2  double y = 4 + 5;
3  double z = 4 + 5.8;
```

Remember that operators of the same priority evaluate left to right. So the first statement is going to be:

```
1  double x = 27 + 4.5;
```

Now lines 1 and 3 have mixed types! Java has two options:

1. Convert all doubles to integers so that all the types are integers or

2. Convert all integers to doubles so that all the types are doubles.

If Java were to choose option 1, it would have to demote which could cause loss of precision. So Java handles this in the only safe manner it can: it promotes integers to doubles before performing arithmetic. Again, we rewrite the code to demonstrate Java's automatic promotion:

```
1  double x = 27.0 + 4.5;
2  double y = 4 + 5;
3  double z = 4.0 + 5.8;
```

Java can now execute the arithmetic operators and our code can be written as:

```
1  double x = 31.5;
2  double y = 9;
3  double z = 9.8;
```

The last piece of the puzzle concerns line 2. We have mismatched types! We have a double typed variable, `y`, but the value to be assigned to it is 9, an `int`. Again, Java will promote the int to a double as that is a safe conversion, and line 2 will become:

```
1  double y = 9.0;
```

Finally! All our types match, and we can clearly see the values that will be assigned to the variables `x`, `y`, and `z` will be `31.5`, `9.0`, and `9.8`.

It seems to be a very complicated process, but remember that there is no ambiguity to a computer program. The program follows rules and if you understand the rules, you can figure out how to think like a computer.

3.5 Overflow errors

Having discussed integers and floating-point numbers, many students think that these are the only types they will ever need to represent numerical values. This isn't the case. Each type has a limit or capacity to the values it can represent.

Think of a type as sort of like a measuring cup or a (non-digital) odometer on a car. You can fill up a measuring cup to the very top, but if you continue to add liquid, the cup will overflow. Likewise, you can drive a car to the point that the odometer rolls over and changes from the maximum number of miles it can represent (99,999.9 in the picture below) to the minimum number of miles it can represent (00000.0 below). This doesn't mean you have a new car! It just means you are at the limits of your technology. In programming, we call these errors **overflow errors**.

Just like measuring cups or odometers, types have a limit to the numerical values they can represent. If we need to store a value larger or smaller then a variable's capacity, we will need to use a different type for that variable.

As programmers, you should be aware that the types have limits and exceeding these limits can lead to an overflow error. Consider the following code:

```
1  int x = 2147483647;  //Maximum value for int type!
2  System.out.println(x + 1);
```

This code will print out the value `-2147483648`! We have created an overflow error! Our "measuring cup" was completely full and we added more to it, causing our overflow error. Or if you prefer, our "odometer" has rolled around from its maximum value and is now showing us the minimum value.

When computer memory was expensive, balancing memory usage against the values which you needed to represent was important. Today, memory is cheap and we can almost always use the `int` type for whole numbers and the `double` type for fractional/decimal numbers. However, if you ever write programs which deal with very large or very small numbers, you will need to carefully consider the limits of your data types.

3.6 Glossary

initialization: A statement that declares a new variable and assigns a value to it at the same time.

floating-point: A type of variable (or value) that can contain fractions as well as integers. The floating-point type we will use is `double`.

overflow error: An error in which we try to represent a value larger or smaller then the capacity of our variable type.

3.7 Exercises

Exercise 3.1.

Chapter 4

Methods

4.1 Math methods

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. Then you can evaluate the function itself, either by looking it up in a table or by performing various calculations.

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

Java provides functions that perform the most common mathematical operations. These functions are called **methods**. The math methods are invoked using a syntax that is similar to the `print` statements we have already seen:

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

The first example sets the variable `root` to the square root of 17. The second example on line 3 finds the sine of the value of `angle`, which is `1.5`. 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 an approximation of π with the constant `Math.PI`:

```
1     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).

4.2 Composition

Just as with mathematical functions, Java methods can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a method:

```
double x = Math.cos(angle + Math.PI/2);
```

This statement takes the value `Math.PI`, divides it by two and adds the result to the value of the variable `angle`. The sum is then passed as an argument to `cos`. (`PI` is the name of a variable, not a method, so there are no arguments, not even the empty argument `()`).

You can also take the result of one method and pass it as an argument to another:

```
double x = Math.exp(Math.log(10.0));
```

In Java, the `log` method always uses base e, so this statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

4.3 Adding new methods

So far we have used methods from Java libraries, but it is also possible to add new methods. We have already seen one method definition: `main`. The method named `main` is special, but the syntax is the same for other methods:

```
public static void NAME( LIST OF PARAMETERS ) {
  STATEMENTS
}
```

The code that appears on line 1 in the above example is something called the **method header**. The method header can tell a programmer all they need to know to be able to use the method in their code. The crucial pieces of the method header are:

1. the type of data (if any) returned by the method. In the above code, this is signified by the keyword `void`. We'll discuss return values more later.

2. the name of the method (ie `NAME` in the code above)

3. the list of parameters the method requires.

You can make up any name you want for your method, except that you can't call it `main` or any Java keyword. By convention, Java methods start with a lower case letter and use "camel caps," just like variable names.

The list of parameters specifies what information, if any, you have to provide to use (or **invoke** or **call**) the new method. Invoking or calling a method causes it to execute.

The parameter for `main` is `String[] args`, which means that whoever invokes `main` has to provide an array of Strings (we'll get to arrays in Chapter 9). The first couple of methods we are going to write have no parameters, so the syntax looks like this:

```
1  public static void newLine() {
2    System.out.println("");
3  }
```

This method is named `newLine`, and the empty parentheses on line 1 mean that it takes no parameters. It contains one statement, which prints an empty `String`, indicated by `""`. Printing a `String` with no letters in it may not seem all that useful, but `println` skips to the next line after it prints, so this statement skips to the next line.

In `main` we invoke this new method the same way we invoke Java methods:

```
1  public static void main(String[] args) {
2    System.out.println("First line.");
3    newLine();
4    System.out.println("Second line.");
5  }
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the lines. What if we wanted more space between the lines? We could invoke the same method repeatedly:

```
public static void main(String[] args) {
  System.out.println("First line.");
  newLine();
  newLine();
  newLine();
  System.out.println("Second line.");
}
```

Or we could write a new method, named `threeLine`, that prints three new lines:

```
public static void threeLine() {
  newLine();  newLine();  newLine();
}

public static void main(String[] args) {
  System.out.println("First line.");
  threeLine();
  System.out.println("Second line.");
}
```

You should notice a few things about this program:

- You can invoke the same method more than once.
- You can have one method invoke another method. In this case, `main` invokes `threeLine` and `threeLine` invokes `newLine`.
- In `threeLine` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). It is usually a good idea to put each statement on its own line, but I sometimes break that rule.

You might wonder why it is worth the trouble to create all these new methods. There are several reasons; this example demonstrates two:

1. Creating a new method gives you an opportunity to give a name to a group of statements. Methods can simplify a program by hiding

a complex computation behind a single statement, and by using English words in place of arcane code. Which is clearer, `newLine` or `System.out.println("")`?

2. Creating a new method can make a program smaller by eliminating repetitive code. For example, to print nine consecutive new lines, you could invoke `threeLine` three times instead of invoking `newLine` nine times.

In Section 7.5 we will come back to this question and list some additional benefits of dividing programs into methods.

4.4 Classes and methods

Pulling together the code fragments from the previous section, the class definition looks like this:

```
1  public class NewLine {
2
3    public static void newLine() {
4      System.out.println("");
5    }
6
7    public static void threeLine() {
8      newLine();  newLine();  newLine();
9    }
10
11   public static void main(String[] args) {
12     System.out.println("First line.");
13     threeLine();
14     System.out.println("Second line.");
15   }
16 }
```

Program 4.1: NewLine.java

The first line indicates that this is the class definition for a new class called `NewLine`. A **class** is a collection of related methods. In this case, the class named `NewLine` contains three methods, named `newLine`, `threeLine`, and `main`.

The other class we've seen is the `Math` class which Java provides for us to use. It contains methods named `sqrt`, `sin`, and many others. When we invoke a mathematical method, we have to specify the name of the class (`Math`)

and the name of the method. That's why the syntax is slightly different for Java methods and the methods we write:

```
1    Math.pow(2.0, 10.0);
2    newLine();
```

The first statement invokes the `pow` method in the `Math class` (which raises the first argument to the power of the second argument). The second statement invokes the `newLine` method, which Java assumes is in the class we are writing (i.e., `NewLine`).

If you try to invoke a method from the wrong class, the compiler will generate an error. For example, if you type:

```
1    pow(2.0, 10.0);
```

The compiler will say something like, "Can't find a method named `pow` in class `NewLine`." If you have seen this message, you might have wondered why it was looking for `pow` in your class definition. Now you know.

4.5 Programs with multiple methods

When you look at a class definition that contains several methods, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `main`, regardless of where it is in the program (in this example I deliberately put it at the bottom). Statements are executed one at a time, in order, until you reach a method invocation. Method invocations are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the invoked method, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one method can invoke another. Thus, while we are in the middle of `main`, we might have to go off and execute the statements in `threeLine`. But while we are executing `threeLine`, we get interrupted three times to go off and execute `newLine`.

For its part, `newLine` invokes `println`, which causes yet another detour. Fortunately, Java is adept at keeping track of where it is, so when `println` completes, it picks up where it left off in `newLine`, and then gets back to `threeLine`, and then finally gets back to `main` so the program can terminate.

Technically, the program does not terminate at the end of `main`. Instead, execution picks up where it left off in the program that invoked `main`, which is the Java interpreter. The interpreter takes care of things like deleting windows and general cleanup, and *then* the program terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This is a skill that is sometimes called **code tracing** because we want to follow, statement-by-statement, what the code is doing.

4.6 Parameters and arguments

Some of the methods we have used require **arguments**, which are values that you provide when you invoke the method. For example, to find the sine of a number, you have to provide the number. So `sin` takes a `double` as an argument. To print a string, you have to provide the string, so `println` takes a `String` as an argument.

Some methods take more than one argument; for example, `pow` takes two `doubles`, the base and the exponent.

When you use a method, you provide arguments. When you write a method, you specify a list of parameters. A **parameter** is a variable that stores an argument. The parameter list indicates what arguments are required.

Let's look at the following program:

```
1  public class Twice {
2     public static void printTwice(String s) {
3        System.out.println(s);
4        System.out.println(s);
5     }
6
7     public static void main(String[] args) {
8        printTwice("I'm getting angry!");
9
10       String argument = "Don't make me say this twice!";
11       printTwice(argument);
12    }
13 }
```

Program 4.2: Twice.java

The method `printTwice` specifies a single parameter, `s`, that has type

`String`. I called it `s` to suggest that it is a `String`, but I could have given it any legal variable name.

When we invoke `printTwice`, we have to provide a single argument with type `String`.

When you invoke a method, the argument(s) you provide is/are assigned to the parameter(s). In this example on line 8, the argument `"I'm getting angry!"` is assigned to the parameter `s`. This processing is called **parameter passing** because the value gets passed from outside the method to the inside.

An argument can be any kind of expression, so if you have a `String` variable, you can use it as an argument. Lines 10-11 demonstrate this concept.

The value you provide as an argument must have the same type as the parameter. For example, if you try this:

```
1     printTwice(17);
```

You get an error message like "cannot find symbol," which isn't very helpful. The reason is that Java is looking for a method named `printTwice` that can take an integer argument. Since there isn't one, it can't find such a "symbol."

Note that the method `System.out.println` can accept any type as an argument. But that is an exception; most methods are not so accommodating.

4.7 Scope and Stack diagrams

Parameters and other variables only exist inside their own methods. This concept is referred to as the **scope** of a variable. Take a look at the following modification of the `Twice.java` program.

```
1  public class TwiceScope {
2     public static void printTwice(String s) {
3        System.out.println(s);
4        System.out.println(s);
5        System.out.println(argument);  //Wrong! argument is not in scope
6     }
7
8     public static void main(String[] args) {
9        String argument = "Don't make me say this twice!";
10       printTwice(argument);
11       System.out.println(s);   //Wrong!  s is not in scope
12    }
13 }
```

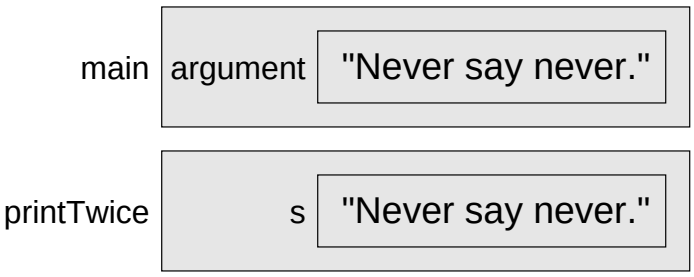
Program 4.3: TwiceScope.java

If you try to compile this program, you will get the following errors from the compiler:

```
TwiceScope.java:5: error: cannot find symbol
  System.out.println(argument);  //Wrong! argument is not in scope
                     ^
  symbol:   variable argument
  location: class TwiceScope
TwiceScope.java:11: error: cannot find symbol
  System.out.println(s);   //Wrong!  s is not in scope
                     ^
  symbol:   variable s
  location: class TwiceScope
2 errors
```

This output tells us there are two errors, and those errors occur on lines 5 and 11 in our TwiceScope.java file. Java even helpfully points to what it thinks are the problems with the caret symbol (^). The error itself, "cannot find symbol" is slightly confusing, but stems from the fact the variables we are trying to use are not in scope. On line 5, Java is looking for the variable argument, but it finds no variable of that name in the current scope (the `printTwice` method).

One way to keep track of where each variable is defined is with a **stack diagram**. The stack diagram for the `TwiceScope` program looks like this:



For each method there is a gray box called a **frame** that contains the method's parameters and variables. The name of the method appears outside the frame. As usual, the value of each variable is drawn inside a box with the name of the variable beside it.

4.8 Methods with multiple parameters

The syntax for declaring and invoking methods with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
    public static void printTime(int hour, int minute) {
        System.out.print(hour);
        System.out.print(":");
        System.out.println(minute);
    }
```

It might be tempting to write:

```
  public static void printTime(int hour, minute)
```

but that format is only legal for variable declarations, not parameter lists.

Another common source of confusion is that you do not have to declare the types of arguments when you invoke a method. The following is wrong!

```
    int hour = 11;
    int minute = 59;
    printTime(int hour, int minute);   // WRONG!
```

In this case, Java can tell the type of `hour` and `minute` by looking at their declarations. It is not necessary to include the type when you pass them as arguments. The correct syntax to invoke the method would be `printTime(hour, minute)`.

4.9 Methods that return values

Some of the methods we are using, like the `Math` methods, return values. Other methods, like `println` and `newLine`, perform an action but they don't return a value. That raises some questions:

- What happens if you invoke a method and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a `print` method as part of an expression, like `System.out.println("boo!") + 7`?
- Can we write methods that return values, or are we stuck with things like `newLine` and `printTwice`?

The answer to the third question is "yes, and we're going to do it now." I leave it up to you to answer the other two questions by trying them out. In fact, any time you have a question about what is legal or illegal in Java, a good way to find out is to ask the compiler.

What do we mean when we say that some functions "produce results?" We mean that the effect of invoking the method is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
double e = Math.exp(1.0);
double height = radius * Math.sin(angle);
```

But so far all our methods have been **void**; that is, methods that return no value. When you invoke a void method, it is typically on a line by itself, with no assignment:

```
printTime(6,30);
newLine();
```

Now we will write methods that return things, which I call **value** methods. The first example is `area`, which takes a `double` as a parameter, and returns the area of a circle with the given radius:

```
public static double area(double radius) {
  double area = Math.PI * radius * radius;
  return area;
}
```

The first thing you should notice is that the beginning of the method definition is different. Instead of `public static void`, which indicates a void method, we see `public static double`, which means that the return value from this method is a `double`. I still haven't explained what `public static` means, but be patient.

The last line is a new form of the `return` statement that includes a return value. This statement means, "return immediately from this method and use the result of the following expression as the return value." The expression you provide can be arbitrarily complicated, so we could have written this method more concisely:

```
public static double area(double radius) {
  return Math.PI * radius * radius;
}
```

On the other hand, **temporary** variables like `area` often make debugging easier. In either case, the type of the expression in the `return` statement must match the return type of the method. In other words, when you declare that the return type is `double`, you are making a promise that this method will eventually produce a `double`. If you try to `return` with no expression, or an expression with the wrong type, the compiler will take you to task.

4.10 Program development

At this point you should be able to look at complete Java methods and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest a method called **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` method should look like in Java. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the parameters, and it is natural to represent them using four `doubles`, although we will see later that there is a `Point` object in Java that we could use. The return value is the distance, which will have type `double`.

Already we can write an outline of the method:

```
public static double distance
            (double x1, double y1, double x2, double y2) {
  return 0.0;
}
```

The statement

```
return 0.0;
```

is a place-holder that is necessary to compile the program. Obviously, at this stage the program doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we add more code.

To test the new method, we have to invoke it with sample values. Somewhere in `main` I would add:

```
   double dist = distance(1.0, 2.0, 4.0, 6.0);
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a method, it is useful to know the right answer.

Once we have checked the syntax of the method definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. If there is an error at any point, we have a good idea where to look: in the last line we added.

The next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. I store those values in temporary variables named `dx` and `dy`.

```
public static double distance
            (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  System.out.println("dx is " + dx);
  System.out.println("dy is " + dy);
  return 0.0;
}
```

I added print statements so we can check the intermediate values before proceeding. They should be 3.0 and 4.0.

When the method is finished I remove the print statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product.

The next step is to square `dx` and `dy`. We could use the `Math.pow` method, but it is simpler to multiply each term by itself.

```
public static double distance
            (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  System.out.println("dsquared is " + dsquared);
  return 0.0;
}
```

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use the `Math.sqrt` method to compute the result and return the value.

```
public static double distance
            (double x1, double y1, double x2, double y2) {
```

```
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    double result = Math.sqrt(dsquared);
    return result;
  }
```

In `main`, we can print and check the value of the result.

As you gain more experience programming, you might write and debug more than one line at a time. Nevertheless, incremental development can save you a lot of time. The key aspects of the process are:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.
- Use temporary variables to hold intermediate values so you can print and check them.
- Once the program is working, you can remove scaffolding and consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

4.11 Composition

Once you define a new method, you can use it as part of an expression, and you can build new methods using existing methods. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a method, `distance` that does that.

```
    double radius = distance(xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius, and return it.

```
    double area = area(radius);
    return area;
```

Wrapping that all up in a method, we get:

```
public static double circleArea
            (double xc, double yc, double xp, double yp) {
  double radius = distance(xc, yc, xp, yp);
  double area = area(radius);
  return area;
}
```

The temporary variables `radius` and `area` are useful for development and debugging, but once the program is working we can make it more concise by composing the method invocations:

```
public static double circleArea
            (double xc, double yc, double xp, double yp) {
  return area(distance(xc, yc, xp, yp));
}
```

4.12 Overloading

You might have noticed that `circleArea` and `area` perform similar functions—finding the area of a circle—but take different parameters. For `area`, we have to provide the radius; for `circleArea` we provide two points.

If two methods do the same thing, it is natural to give them the same name. Having more than one method with the same name, which is called **overloading**, is legal in Java *as long as each version takes different parameters*. So we could rename `circleArea`:

```
public static double area
            (double x1, double y1, double x2, double y2) {
  return area(distance(xc, yc, xp, yp));
}
```

When you invoke an overloaded method, Java knows which version you want by looking at the arguments that you provide. If you write:

```
  double x = area(3.0);
```

Java goes looking for a method named `area` that takes one `double` as an argument, and so it uses the first version, which interprets the argument as a radius. If you write:

```
  double x = area(1.0, 2.0, 4.0, 6.0);
```

Java uses the second version of `area`. And notice that the second version of

`area` actually invokes the first.

Many Java methods are overloaded, meaning that there are different versions that accept different numbers or types of parameters. For example, there are versions of `print` and `println` that accept a single parameter of any type. In the Math class, there is a version of `abs` that works on `double`s, and there is also a version for `int`s.

Although overloading is a useful feature, it should be used with caution. You might get yourself nicely confused if you are trying to debug one version of a method while accidently invoking a different one.

And that reminds me of one of the cardinal rules of debugging: **make sure that the version of the program you are looking at is the version of the program that is running!**

Some day you may find yourself making one change after another in your program, and seeing the same thing every time you run it. This is a warning sign that you are not running the version of the program you think you are. To check, add a `print` statement (it doesn't matter what you print) and make sure the behavior of the program changes accordingly.

4.13 Glossary

initialization: A statement that declares a new variable and assigns a value to it at the same time.

floating-point: A type of variable (or value) that can contain fractions as well as integers. The floating-point type we will use is `double`.

overflow error: An error in which we try to represent a value larger or smaller then the capacity of our variable type.

class: A named collection of methods. So far, we have used the `Math` class and the `System` class, and we have written classes named `Hello` and `NewLine`.

method: A named sequence of statements that performs a useful function. Methods may or may not take parameters, and may or may not return a value.

parameter: A piece of information a method requires before it can run. Parameters are variables: they contain values and have types.

method header: A single line of code (usually the first line of a method) which specifies the return type, name, and parameters of a method.

argument: A value that you provide when you invoke a method. This value must have the same type as the corresponding parameter.

frame: A structure (represented by a gray box in stack diagrams) that contains a method's parameters and variables.

invoke: Cause a method to execute. This concept may also be referred to as "calling" a method.

scope: The portion of a program where it is valid to use a particular variable.

return type: The part of a method declaration that indicates what type of value the method returns.

return value: The value provided as the result of a method invocation.

dead code: Part of a program that can never be executed, often because it appears after a `return` statement.

scaffolding: Code that is used during program development but is not part of the final version.

void: A special return type that indicates a void method; that is, one that does not return a value.

overloading: Having more than one method with the same name but different parameters. When you invoke an overloaded method, Java knows which version to use by looking at the arguments you provide.

4.14 Exercises

Exercise 4.1. Draw a stack frame that shows the state of the program in Section 4.8 when `main` invokes `printTime` with the arguments `11` and `59`.

Exercise 4.2. The point of this exercise is to practice reading code and to make sure that you understand the flow of execution through a program with multiple methods.

1. What is the output of the following program? Be precise about where there are spaces and where there are newlines.

   HINT: Start by describing in words what `ping` and `baffle` do when they are invoked.

2. Draw a stack diagram that shows the state of the program the first time `ping` is invoked.

```
public static void zoop() {
  baffle();
  System.out.print("You wugga ");
  baffle();
}

public static void main(String[] args) {
  System.out.print("No, I ");
  zoop();
  System.out.print("I ");
  baffle();
}

public static void baffle() {
  System.out.print("wug");
  ping();
}

public static void ping() {
  System.out.println(".");
}
```

Exercise 4.3. The point of this exercise is to make sure you understand how to write and invoke methods that take parameters.

1. Write the method header of a method named `zool` that takes three parameters: an `int` and two `Strings`.

2. Write a line of code that invokes `zool`, passing as arguments the value `11`, the name of your first pet, and the name of the street you grew up on.

Exercise 4.4. The purpose of this exercise is to take code from a previous exercise and encapsulate it in a method that takes parameters. You should start with a working solution to Exercise 2.3.

1. Write a method called `printAmerican` that takes the day, date, month and year as parameters and that prints them in American format.

2. Test your method by invoking it from `main` and passing appropriate arguments. The output should look something like this (except that the date might be different):

   ```
   Saturday, July 16, 2011
   ```

3. Once you have debugged `printAmerican`, write another method called `printEuropean` that prints the date in European format.

Exercise 4.5. Many computations can be expressed concisely using the "multadd" operation, which takes three operands and computes `a*b + c`. Some processors even provide a hardware implementation of this operation for floating-point numbers.

1. Create a new program called `Multadd.java`.

2. Write a method called `multadd` that takes three `doubles` as parameters and that returns their multadditionization.

3. Write a `main` method that tests `multadd` by invoking it with a few simple parameters, like `1.0, 2.0, 3.0`.

4. Also in `main`, use `multadd` to compute the following values:

$$\sin \tfrac{\pi}{4} + \frac{\cos \frac{\pi}{4}}{2}$$
$$\log 10 + \log 20$$

5. Write a method called `yikes` that takes a double as a parameter and that uses `multadd` to calculate

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

   HINT: the Math method for raising e to a power is `Math.exp`.

In the last part, you get a chance to write a method that invokes a method you wrote. Whenever you do that, it is a good idea to test the first method carefully before you start working on the second. Otherwise, you might find yourself debugging two methods at the same time, which can be difficult.

One of the purposes of this exercise is to practice pattern-matching: the ability to recognize a specific problem as an instance of a general category of problems.

Chapter 5

Conditionals and booleans

5.1 The modulo operator

The modulo operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In Java, the modulo operator (often just called "mod") is a percent sign, `%`. The syntax is 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 modulo 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 modulo 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.

5.2 Conditional execution

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

```
public class SimpleIf {
```

```
2   public static void main(String[] args) {
3     if (x > 0) {
4       System.out.println("x is positive");
5     }
6   }
7 }
```

Program 5.1: SimpleIf.java

The expression inside parentheses on line 3 is called the condition. If it is true, then the statements inside the curly braces get executed and the program prints `x is positive`. If the condition is not true, the statements inside the curly braces of the if statement are skipped, nothing happens and the program terminates.

The condition can contain any of the comparison operators, sometimes called **relational operators**:

```
1     x == y               // x equals y
2     x != y               // x is not equal to y
3     x > y                // x is greater than y
4     x < y                // x is less than y
5     x >= y               // x is greater than or equal to y
6     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 reverse the symbols in an operator: there is no such thing as `=<` or `=>`.

It is especially important to distinguish between an assignment statement and a statement of equality. Because Java uses the `=` symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First of all, equality is commutative, and assignment is not. For example, in mathematics if $a = 7$ then $7 = a$. But in Java `a = 7;` is a legal assignment statement which assigns the value `7` to the variable `a`, and `7 = a;` is not.

Furthermore, 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!

```
1     int a = 5;
2     int b = a;      // assign the value of b to a too!
3                     // a and b are now equal
4     a = 3;          // a and b are no longer equal
```

Line 4 changes the value of `a` but it does not change the value of `b`, so they are no longer equal. In some programming languages a different symbol is used for assignment, such as `<-` or `:=`, to avoid this confusion.

When using comparison operators, you must also be careful about type. The two sides of a comparison operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`. (But remember, Java may automatically convert types for you as long as it is safe to do so!)

The operators `==` and `!=` work with Strings, but they don't do what you expect. And the other relational operators won't even compile if used with strings. We will see how to compare strings Section 8.12. For now, assume you can use these operators only with numerical types.

5.3 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:

```
1 public class SimpleIfElse {
2   public static void main(String[] args) {
3     if (x % 2 == 0) {
4       System.out.println("x is even");
5     } else {
6       System.out.println("x is odd");
7     }
8   }
9 }
```

Program 5.2: SimpleIfElse.java

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code prints a message to that effect. If the condition is false, the second print statement is executed (line 6). Since the condition must be true or false, exactly one of the `println` statements (line 4 or line 6) will be executed. This program will always print exactly one thing. It cannot print nothing, and it cannot print both "x is even" and "x is odd."

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a method, as follows:

```
1   public static void printParity(int x) {
```

```
2     if (x % 2 == 0) {
3       System.out.println("x is even");
4     } else {
5       System.out.println("x is odd");
6     }
7   }
```

Now you have a method named `printParity` that will print an appropriate message for any integer you care to provide. Java will always execute *either* line 3 or line 5. It will never execute both and it will never execute neither.

In `main` you would invoke this method like this:

```
1     printParity(17);
```

Always remember that when you *invoke* a method, you do not have to declare the types of the arguments you provide. Java can figure out what type they are. You should resist the temptation to write things like:

```
1     int number = 17;
2     printParity(int number);          // WRONG!!!
```

5.4 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:

```
1  public class ChainingIf {
2     public static void main(String[] args) {
3        if (x > 0) {
4        System.out.println("x is positive");
5      } else if (x < 0) {
6        System.out.println("x is negative");
7      } else {
8        System.out.println("x is zero");
9        }
10    }
11 }
```

Program 5.3: ChainingIf.java

In this code, exactly one (and only one!) of the `println` statements (found on lines 4, 6, and 8) will execute. The code will never print nothing and the code will never print more than one statement. This makes sense as

a number cannot be both positive and negative or both negative and zero, etc.

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 curly-braces lined up, you are less likely to make syntax errors and more likely to find them if you do.

5.5 Overlapping conditions

In the previous section, we discussed how we can chain conditional statements together and only one of them will execute. However, we can write code in which the conditions logically overlap, and it's important to understand the flow of execution the computer takes. Take a look at this example:

```
1    int number = 20;
2    if (number % 2 == 0) {
3       System.out.println("Your number is even!");
4    } else if (number % 10 == 0) {
5       System.out.println("Your number is divisible by 10!");
6    } else {
7      System.out.println("Your number isn't special.");
8    }
9    System.out.println("Finished!");
```

In this piece of code, the variable `number` is assigned the value of 20. After that, the computer begins by testing our first conditional: `number % 2 == 0`. This tests if `number` is even. Since the remainder is 0 when `number` is divided by 2, we will execute the print statement on line 3. At that point, execution will skip the remainder of our chained conditionals (including the `else` portion) and execute the code on line 9.

It's tempting for students to evaluate each and every conditional and include the output generated by the `println` on line 5 because 20 is also divisible by 10 and thus the condition on line 4 is true. However, that's not how Java works in this instance. As soon as a condition evaluates to true, Java will execute the block of code and then skip the remaining chained conditionals. Because of this flow of execution, the totality of the code on lines 2-7 is often referred to simply as an if-statement. Java treats the entire set as one unit of code to be evaluated and executed.

5.6 Nested conditionals

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

```
1   if (x == 0) {
2     System.out.println("x is zero");
3   } else {
4     if (x > 0) {
5       System.out.println("x is positive");
6     } else {
7       System.out.println("x is negative");
8     }
9   }
```

We refer to the if statement which begins on line 1 as an **outer conditional** and the if statement which begins on line 4 as an **inner conditional**.

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. Those two branches are both `print` statements, but they could have been conditional statements as well.

Indentation helps make the structure apparent, but nevertheless, nested conditionals can be difficult to read very quickly. A good rule of thumb is to try to avoid a nested conditional that is more than "two levels" deep.

This sort of **nested structure** is common, and we will see it again later.

5.7 The return statement

In the previous chapter, we saw how we could use a `return` statement to make our methods produce a result (value) for us. However, return statements have another trait which effects the flow of execution.

The `return` statement also allows you to terminate the execution of a method before you reach the end. As soon as the flow of execution encounters a `return` statement, execution of the method ceases and the execution continues at the point the method was invoked.

In programming parlance, we say that a method **returns** when it completes. So we now have two different ways a function can finish:

1. By completing all the statements in the body of the method.

2. By encountering a `return <expression>;` statement during the execution of the method. (For value methods.)

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
public static double absoluteValue(double x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Since these return statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one return statement in a method, you should keep in mind that as soon as one is executed, the method terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**. Some compilers warn you if part of your code is dead.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program hits a return statement. For example:

```
public static double absoluteValue(double x) {
  if (x < 0) {
    return -x;
  } else if (x > 0) {
    return x;
  }                          // WRONG!!
}
```

As can be seen in our method header, this method *must* return a `double` value. This program is not legal because if `x` is 0, neither condition is true and the method ends without returning a value. A typical compiler message would be "return statement required in absoluteValue," which is a confusing message since there are already two `return` statements in our method.

Now, we'll add one more use: a `return` statement in a `void` method to end the execution of a method which does not have a return value. The syntax is a very simple statement:

```
  return;
```

Note that this `return` statement does not have a value or expression after it. It is not causing our method to return a value; it is just ending the method and the flow of execution will then continue at the point at which the method was invoked.

One reason to use it is if you detect an error condition. Take a look at the `printLogarthim` method in this program:

```
1  public class BasicReturn {
2    public static void printLogarithm(double x) {
3      if (x <= 0.0) {
4        System.out.println("Positive numbers only, please.");
5        return;
6      }
7
8      double result = Math.log(x);
9      System.out.println("The log of x is " + result);
10   }
11
12    public static void main(String[] args) {
13     printLogarithm(15.0);
14     printLogarithm(-15.0);
15   }
16 }
```

Program 5.4: BasicReturn.java

The `printLogarithm` method takes a `double` named `x` as a parameter. It checks whether `x` is less than or equal to zero, in which case it prints an error message and then uses `return` to exit the method. The flow of execution immediately returns to the point from which is was called and the remaining lines of the method (lines 7-10) are not executed.

I used a floating-point value on the right side of the condition on line 3 because there is a floating-point variable on the left.

The `main` method invokes this method twice, once with a positive number and once with a negative number to demonstrate how the `return` statement works.

5.8 Type conversion

On line 9 in the `BasicReturn.java` program above, you might wonder how you can get away with an expression like `"The log of x is " + result`, since one of the operands is a `String` and the other is a `double`. In this case

Java is being smart on our behalf, automatically converting the `double` to a `String` before it does the string concatenation.

Whenever you use the `+` operator, if one of the operands is a `String`, Java converts the other to a `String` and then performs string concatenation. What do you think happens if you use the `+` operator with an integer and a floating-point value as operands?

5.9 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 **relational operators**, which compare numerical values and return either `true` or `false`. `true` and `false` are special values in Java, and together they make up a type called **boolean**. You might recall that when I defined a type, I said it was a set of values. In the case of `int`s, `double`s and `String`s, those sets are pretty big. For `boolean`s, there are only two values.

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

```
1    boolean flag;
2    flag = true;
3    boolean testResult = false;
```

Line 1 is a simple variable declaration; line 2 is an assignment statement, and line 3 is an initialization.

The values `true` and `false` are keywords in Java, so they may appear in a different color, depending on your development environment.

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

```
1    boolean evenFlag = (n % 2 == 0);      // true if n is even
2    boolean positiveFlag = (x > 0);       // true if x is positive
```

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

```
1    if (evenFlag) {
2      System.out.println("n was even when I checked it");
3    }
```

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

5.10 Logical operators

Consider the situation where you want to figure out if a variable's value is between to other values. For example, what if we want to know if `x` is greater than 0 but less than 10? A lot of students want try this code:

```
int x = 5;
if (0 < x < 10) {
  System.out.println("x is between 0 and 10");
}
```

However, this yields a cryptic error when compiled:

```
error: bad operand types for binary operator '<'
  if(0 < x < 10) {
           ^
first type:  boolean
second type: int
```

This is really confusing! We didn't mix our operand types, but Java appears to be complaining about operand types. We'll return to a detailed explanation of this error in a minute, but for now, let's talk about how to fix it.

To fix our problem, we need to break it down into smaller pieces. We need to know if `x` is greater than 0 *and* if `x` is less than 10.

There are three **logical operators** in Java: AND, OR and NOT, which are denoted by the symbols `&&`, `||` and `!`. The semantics 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.

`evenFlag || n%3 == 0` is true if *either* of the conditions is true, that is, if `evenFlag` is true OR the number is divisible by 3.

Finally, the NOT operator inverts a boolean expression, so `!evenFlag` is true if `evenFlag` is false—if the number is odd.

Now, let's understand the error we had earlier. Why would the conditional test `0 < x < 10` cause Java to complain about bad operand types? To understand, we need to remember what happens when we have 2 operators of the same priority. In this case the two comparison operators are the same (`<`) and so our expression is evaluated left to right. Java first encounters `0`

`< x` and evaluates it to `true`. Then Java tries to complete the rest of the comparison by substituting the value `true`: `true < 10`. Uh-oh! Mismatched types and Java can't handle this. Note in our error message (lines 4 and 5) that Java explicitly says the first operand is a `boolean` and the second operand is an `int`.

Logical operators can simplify nested conditional statements. For example, can you re-write this code using a single conditional?

```
1    if (x >= 13) {
2      if (x < 20) {
3        System.out.println("x is a teenager");
4      }
5    }
```

5.11 A note on style

As you might have noticed, there are numerous syntactically correct ways to write code. For the previous example of a "teenager" number, we could write any of the below:

```
1    if (x >= 13) {
2      if (x < 20) {
3        System.out.println("x is a teenager");
4      }
5    }
```

```
1    if (x >= 13 && x < 20) {
2        System.out.println("x is a teenager");
3    }
```

```
1    if (x > 12) {
2      if (x < 20) {
3        System.out.println("x is a teenager");
4      }
5    }
```

```
1    if (x > 12 && x < 20) {
2        System.out.println("x is a teenager");
3    }
```

```
1    if (x >= 13 && x <= 19) {
2        System.out.println("x is a teenager");
3    }
```

```
    if (x <= 19) {
      if (x >= 13) {
        System.out.println("x is a teenager");
      }
    }
```

and on and on and on. None of these are incorrect. They will all compile and execute correctly. Just like novelists or singers, programmers develop their own style which mimics how they think and solve problems. Don't automatically assume your work is incorrect just because your friend, a TA, or even the professor solves the problem a different way.

5.12 Boolean methods

Methods can return boolean values just like any other type, which is often convenient for hiding tests inside methods. For example:

```
public static boolean isSingleDigit(int x) {
  if (x >= 0 && x < 10) {
    return true;
  } else {
    return false;
  }
}
```

The name of this method is `isSingleDigit`. It is common to give boolean methods names that sound like yes/no questions. The return type is `boolean`, which means that every return statement has to provide a boolean expression.

The code itself is straightforward, although it is longer than it needs to be. Remember that the expression `x >= 0 && x < 10` has type boolean, so there is nothing wrong with returning it directly and avoiding the `if` statement altogether:

```
public static boolean isSingleDigit(int x) {
  return (x >= 0 && x < 10);
}
```

In `main` you can invoke this method in the usual ways:

```
boolean bigFlag = !isSingleDigit(17);
System.out.println(isSingleDigit(2));
```

The first line sets the variable `bigFlag` to `true` only if 17 is *not* a single-digit number. The second line prints `true` because 2 is a single-digit number.

The most common use of boolean methods is inside conditional statements

```
if (isSingleDigit(x)) {
  System.out.println("x is little");
} else {
  System.out.println("x is big");
}
```

5.13 Glossary

modulo: An operator that works on integers and yields the remainder when one number is divided by another. In Java it is denoted with a percent sign(`%`).

conditional: A block of statements that may or may not be executed depending on some condition.

chaining: A way of joining several conditional statements in sequence.

nesting: Putting a conditional statement inside one or both branches of another conditional statement.

casting: Use of an operator that converts from one type to another. In Java it appears as a type name in parentheses, like `(int)` or `(double)`.

boolean: A type of variable that can contain only the two values `true` and `false`.

flag: A variable (usually `boolean`) that records a condition or status information.

conditional operator: An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator: An operator that combines boolean values and produces boolean values.

5.14 Exercises

Exercise 5.1. This exercise reviews the flow of execution through a program with multiple methods. Read the following code and answer the questions below.

```
public class Buzz {

    public static void baffle(String blimp) {
        System.out.println(blimp);
        zippo("ping", -5);
    }

    public static void zippo(String quince, int flag) {
        if (flag < 0) {
            System.out.println(quince + " zoop");
        } else {
            System.out.println("ik");
            baffle(quince);
            System.out.println("boo-wa-ha-ha");
        }
    }

    public static void main(String[] args) {
        zippo("rattle", 13);
    }
}
```

1. Write the number `1` next to the first *statement* of this program that will be executed. Be careful to distinguish things that are statements from things that are not.

2. Write the number `2` next to the second statement, and so on until the end of the program. If a statement is executed more than once, it might end up with more than one number next to it.

3. What is the value of the parameter `blimp` when `baffle` gets invoked?

4. What is the output of this program?

Exercise 5.2. What is the output of the following program?

```
public class Narf {

```

```
    public static void zoop(String fred, int bob) {
        System.out.println(fred);
        if (bob == 5) {
            ping("not ");
        } else {
            System.out.println("!");
        }
    }

    public static void main(String[] args) {
        int bizz = 5;
        int buzz = 2;
        zoop("just for", bizz);
        clink(2*buzz);
    }

    public static void clink(int fork) {
        System.out.print("It's ");
        zoop("breakfast ", fork) ;
    }

    public static void ping(String strangStrung) {
        System.out.println("any " + strangStrung + "more ");
    }
}
```

Exercise 5.3. Fermat's Last Theorem says that there are no integers a, b, and c such that

$$a^n + b^n = c^n$$

except in the case when $n = 2$.

Write a method named `checkFermat` that takes four integers as parameters—`a`, `b`, `c` and `n`—and that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that $a^n + b^n = c^n$, the program should print "Holy smokes, Fermat was wrong!" Otherwise the program should print "No, that doesn't work."

You should assume that there is a method named `raiseToPow` that takes two integers as arguments and that raises the first argument to the power of the second. For example:

```
    int x = raiseToPow(2, 3);
```

would assign the value `8` to `x`, because $2^3 = 8$.

Exercise 5.4. Think about the following problem description:

You want to write a method which returns the price for a haircut. But the cost is different, depending on two different factors: the age of the patron and whether they are male or female. A man's haircut costs $12 but a woman's haircut costs $30. If a child is less than 12 years old, they get a discount. The cost of a boy's haircut is $7 and a girl's haircut is $15.

You want to write a method named `haircutPrice` which takes 2 arguments: the age of the patron and their sex. For simplicity's sake, you can represent the sex as an `int`: 0 for male and 1 for female. The method should return the cost of the haircut as specified above.

- Write the `haircutPrice` method using nested conditional statements.
- Write the `haircutPrice` method using logical operators to form complex conditions in a single chained conditional statement.

Exercise 5.5. Write a method named `isDivisible` that takes two integers, `n` and `m` and that returns `true` if `n` is divisible by `m` and `false` otherwise.

Exercise 5.6. If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

> "If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can."

Write a method named `isTriangle` that it takes three integers as arguments, and that returns either `true` or `false`, depending on whether you can or cannot form a triangle from sticks with the given lengths.

The point of this exercise is to use conditional statements to write a value method.

Exercise 5.7. What is the output of the following program? The purpose of this exercise is to make sure you understand logical operators and the flow of execution through value methods.

```
public static void main(String[] args) {
    boolean flag1 = isHoopy(202);
    boolean flag2 = isFrabjuous(202);
    System.out.println(flag1);
    System.out.println(flag2);
    if (flag1 && flag2) {
        System.out.println("ping!");
    }
    if (flag1 || flag2) {
        System.out.println("pong!");
    }
}

public static boolean isHoopy(int x) {
    boolean hoopyFlag;
    if (x%2 == 0) {
        hoopyFlag = true;
    } else {
        hoopyFlag = false;
    }
    return hoopyFlag;
}

public static boolean isFrabjuous(int x) {
    boolean frabjuousFlag;
    if (x > 0) {
        frabjuousFlag = true;
    } else {
        frabjuousFlag = false;
    }
    return frabjuousFlag;
}
```

Exercise 5.8. (This exercise is based on page 44 of Ableson and Sussman's *Structure and Interpretation of Computer Programs*.)

The following technique is known as Euclid's Algorithm because it appears in Euclid's *Elements* (Book 7, ca. 300 BC). It may be the oldest nontrivial algorithm[1].

The process is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are the same as the

[1]For a definition of "algorithm", jump ahead to Section 11.13.

common divisors of b and r. Thus we can use the equation

$$gcd(a, b) = gcd(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$$gcd(36, 20) = gcd(20, 16) = gcd(16, 4) = gcd(4, 0) = 4$$

implies that the GCD of 36 and 20 is 4. It can be shown that for any two starting numbers, this repeated reduction eventually produces a pair where the second number is 0. Then the GCD is the other number in the pair.

Write a method called `gcd` that takes two integer parameters and that uses Euclid's algorithm to compute and return the greatest common divisor of the two numbers.

Chapter 6

Recursion

6.1 Recursion

I mentioned in the last chapter that it is legal for one method to invoke another, and we have seen several examples. 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 static 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 this method, in `main`, 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 in `main`. So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the methods `newLine` and `threeLine`.

```
1  public static void newLine() {
2    System.out.println("");
3  }
4
5  public static void threeLine() {
6    newLine();  newLine();  newLine();
7  }
```

Although these work, they would not be much help if we wanted to print 2 newlines, or 106. A better alternative would be

```
1  public static void nLines(int n) {
2    if (n > 0) {
3      System.out.println("");
4      nLines(n-1);
5    }
6  }
```

This program similar to `countdown`; as long as `n` is greater than zero, it prints a newline and then invokes itself to print `n-1` additional newlines. The total number of newlines that get printed is `1 +(n-1)`, which usually comes out to roughly `n`.

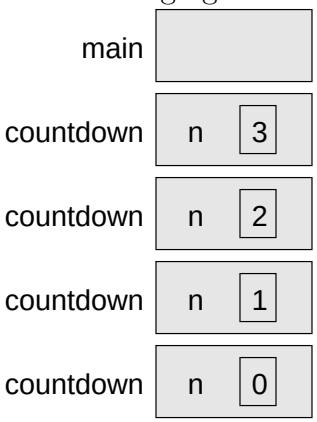
When a method invokes itself, that's called **recursion**, and such methods are **recursive**.

6.2 Stack diagrams for recursive methods

In the previous chapter we used a stack diagram to represent the state of a program during a method invocation. The same kind of diagram can make it easier to interpret a recursive method.

Remember that every time a method gets called it creates a new frame that contains a new version of the method's parameters and variables.

The following figure is a stack diagram for countdown, called with `n = 3`:



There is one frame for `main` and four frames for `countdown`, each with a different value for the parameter `n`. The bottom of the stack, `countdown` with `n=0` is called the **base case**. It does not make a recursive call, so there are no more frames for `countdown`.

The frame for `main` is empty because `main` does not have any parameters or variables.

6.3 More recursion

Now that we have methods that return values, we have a **Turing complete** programming language, which means that we can compute anything com-

putable, for any reasonable definition of "computable." This idea was developed by Alonzo Church and Alan Turing, so it is known as the Church-Turing thesis. You can read more about it at http://en.wikipedia.org/wiki/Turing_thesis.

To give you an idea of what you can do with the tools we have learned, let's look at some methods for evaluating recursively-defined mathematical functions. 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 not very useful:

recursive: an adjective used to describe a method that is recursive.

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 symbol !, which is not to be confused with the 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.

If you can write a recursive definition of something, you can usually write a Java method to evaluate it. The first step is to decide what the parameters are and what the return type is. Since factorial is defined for integers, the method takes an integer as a parameter and returns an integer:

```
1 public static int factorial(int n) {
2 }
```

If the argument happens to be zero, return 1:

```
1 public static int factorial(int n) {
2   if (n == 0) {
3     return 1;
4   }
5 }
```

That's the base case.

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

```
1  public static int factorial(int n) {
2    if (n == 0) {
3      return 1;
4    } else {
5      int recurse = factorial(n-1);
6      int result = n * recurse;
7      return result;
8    }
9  }
```

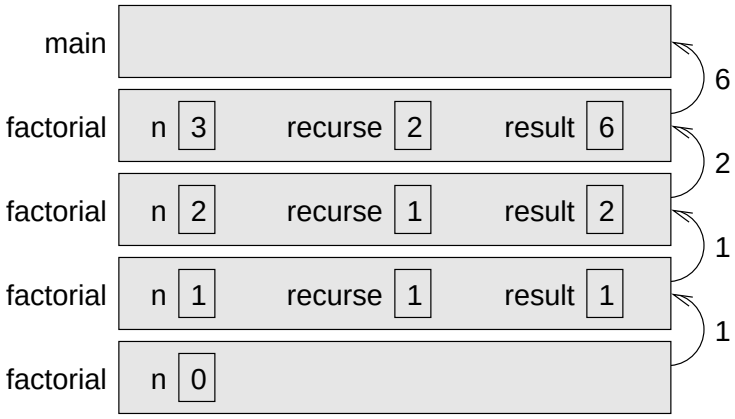
The flow of execution for this program is similar to `countdown` from Section 6.1. If we invoke `factorial` with the value 3:

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 `main`, or whoever invoked `factorial(3)`.

Here is what the stack diagram looks like for this sequence of method invocations:



The return values are shown being passed back up the stack.

Notice that in the last frame `recurse` and `result` do not exist because when `n=0` the branch that creates them does not execute.

6.4 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become disorienting. 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 Java methods. When you invoke `Math.cos` or `System.out.println`, you don't examine the implementations of those methods. You just assume that they work.

You can apply the same logic to your own methods. For example, in Section 5.12 we wrote a method called `isSingleDigit` that determines whether a number is between 0 and 9. Once we convince ourselves that this method is correct—by testing and examination of the code—we can use the method without ever looking at the code again.

The same is true of recursive programs. When you get to the recursive invocation, instead of following the flow of execution, you should *assume* that the recursive invocation works, and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" Yes, you can, by multiplying by n.

Of course, it is 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!

6.5 One more example

The second most common 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

```
1  public static int fibonacci(int n) {
2    if (n == 0 || n == 1) {
3      return 1;
4    } else {
5      return fibonacci(n-1) + fibonacci(n-2);
6    }
7  }
```

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

6.6 Glossary

recursion: The process of invoking the same method you are currently executing.

base case: A condition that causes a recursive method *not* to make a recursive call.

6.7 Exercises

Exercise 6.1. Draw a stack diagram that shows the state of the program in Section 6.1 after `main` invokes `nLines` with the parameter `n=4`, just before the last invocation of `nLines` returns.

Exercise 6.2. The first verse of the song "99 Bottles of Beer" is:

> 99 bottles of beer on the wall, 99 bottles of beer, ya' take one down, ya' pass it around, 98 bottles of beer on the wall.

Subsequent verses are identical except that the number of bottles gets smaller by one in each verse, until the last verse:

> No bottles of beer on the wall, no bottles of beer, ya' can't take one down, ya' can't pass it around, 'cause there are no more bottles of beer on the wall!

And then the song(finally) ends.

Write a program that prints the entire lyrics of "99 Bottles of Beer." Your program should include a recursive method that does the hard part, but you might want to write additional methods to separate the major functions of the program.

As you develop your code, test it with a small number of verses, like "3 Bottles of Beer."

The purpose of this exercise is to take a problem and break it into smaller problems, and to solve the smaller problems by writing simple methods.

Exercise 6.3. The point of this exercise is to use a stack diagram to understand the execution of a recursive program.

```
1  public class Prod {
2
3      public static void main(String[] args) {
4          System.out.println(prod(1, 4));
5      }
6
7      public static int prod(int m, int n) {
8          if (m == n) {
9              return n;
10         } else {
11             int recurse = prod(m, n-1);
12             int result = n * recurse;
13             return result;
14         }
15     }
16 }
```

1. Draw a stack diagram showing the state of the program just before the last instance of `prod` completes. What is the output of this program?

2. Explain in a few words what `prod` does.

3. Rewrite `prod` without using the temporary variables `recurse` and `result`.

Exercise 6.4. The purpose of this exercise is to translate a recursive definition into a Java method. The Ackerman function is defined for non-negative integers as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases} \tag{6.1}$$

Write a method called `ack` that takes two `int`s as parameters and that computes and returns the value of the Ackerman function.

Test your implementation of Ackerman by invoking it from `main` and printing the return value.

WARNING: the return value gets very big very quickly. You should try it only for small values of m and n (not bigger than 2).

Exercise 6.5. 1. Create a program called `Recurse.java` and type in the following methods:

```
    // first: returns the first character of the given String
    public static char first(String s) {
        return s.charAt(0);
    }

    // last: returns a new String that contains all but the
    // first letter of the given String
    public static String rest(String s) {
        return s.substring(1, s.length());
    }

    // length: returns the length of the given String
    public static int length(String s) {
        return s.length();
    }
```

2. Write some code in `main` that tests each of these methods. Make sure they work, and make sure you understand what they do.

3. Write a method called `printString` that takes a String as a parameter and that prints the letters of the String, one on each line. It should be a `void` method.

4. Write a method called `printBackward` that does the same thing as `printString` but that prints the String backward (one character per line).

5. Write a method called `reverseString` that takes a String as a parameter and that returns a new String as a return value. The new String should contain the same letters as the parameter, but in reverse order. For example, the output of the following code

```
1    String backwards = reverseString("Allen Downey");
2    System.out.println(backwards);
```

should be

```
yenwoD nellA
```

Exercise 6.6. Write a recursive method called `power` that takes a double `x` and an integer `n` and that returns x^n.

Hint: a recursive definition of this operation is $x^n = x \cdot x^{n-1}$. Also, remember that anything raised to the zeroeth power is 1.

Optional challenge: you can make this method more efficient, when `n` is even, using $x^n = \left(x^{n/2}\right)^2$.

Chapter 7

Iteration and loops

7.1 The `while` statement

Computers are often used to automate repetitive tasks. Repeating tasks without making errors is something that computers do well and people do poorly.

We have already seen methods like `countdown` and `factorial` that use recursion to perform repetition. This process is also called **iteration**. Java provides language features that make it easier to write these methods. In this chapter we are going to look at the `while` statement. Later on (in Section 9.5) will check out the `for` statement.

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

```
1  public static void countdown(int n) {
2    while (n > 0) {
3      System.out.println(n);
4      n = n-1;
5    }
6    System.out.println("Blastoff!");
7  }
```

You can almost read a `while` statement like English. What this means is, "While `n` is greater than zero, print the value of `n` and then reduce 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 boolean expression (condition) in parentheses on line 2, yielding either `true` or `false`.

2. If the condition evaluates to `false`, exit the `while` statement and continue execution at the next statement on line 6.

3. If the condition evaluates to `true`, execute the statements between the curly-braces (lines 3 and 4), and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to step 1. The statements inside the loop are called the **body** of the loop which is defined by the curly braces on lines 2 and 5. We often talk about whether or not the body of the `while` statement will be executed. The condition inside the parentheses is really a boolean expression. And a boolean expression should evaluate to a `boolean` value: `true` or `false`. If the condition evaluates to `false` the first time it is tested, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that, eventually, the condition evaluates to `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 terminates if `n` is positive. In other cases it is not so easy to tell, as in the case of the method `sequence` in this program:

```
1  public class Collatz {
2    public static void sequence(int n) {
3      while (n != 1) {
4        System.out.println(n);
5        if (n%2 == 0) {           // n is even
6          n = n / 2;
7        } else {                  // n is odd
8          n = n*3 + 1;
9        }
10     }
11   }
12
13   public static void main(String[] args) {
14     sequence(3);
15   }
16 }
```

Program 7.1: Collatz.java

The condition for the loop in the method `sequence` 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 $3n + 1$. In the method `main`, 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. The previous example ends with such a sequence, starting with 16.

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! For more information, see
http://en.wikipedia.org/wiki/Collatz_conjecture.

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 results were 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 were so pervasive that the tables became obsolete.

Well, almost. 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[1].

[1] See http://en.wikipedia.org/wiki/Pentium_FDIV_bug.

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 code 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` method uses?

Since powers of two are important in computer science, we often want logarithms with respect to base 2. To compute them, we can use the formula:

$$\log_2 x = log_e x / log_e 2$$

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 code 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! If you plan to continue in computer science you should memorize the powers of two up to 65536 (that's 2^{16}). I promise you it will come in handy.

7.3 Two-dimensional tables

A two-dimensional table consists of rows and columns that make it easy to find values at the intersections. A multiplication table is a good example. Let's say you want 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; when `i` is 7, the loop terminates. Each time through the loop, we print the value `2*i` and three spaces. Since we use `System.out.print`, 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 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. We have seen two examples of encapsulation, when we wrote `printParity` in Section 5.3 and `isSingleDigit` in Section 5.12.

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`.

```
1  public static void printMultiples(int n) {
2    int i = 1;
3    while (i <= 6) {
4      System.out.print(n*i + "   ");
5      i = i + 1;
6    }
7    System.out.println("");
8  }
```

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.

```
1    int i = 1;
2    while (i <= 6) {
3      printMultiples(i);
4      i = i + 1;
5    }
```

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 and encapsulation

In Section 4.3 I listed some of the reasons methods are useful. Here are several more:

- 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.

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

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

The development process I am demonstrating is called **encapsulation and generalization**. You start by adding code to `main` or another method. When you get it working, you extract it and wrap it up in a method. Then you generalize the method by adding parameters.

Sometimes you don't know when you start writing exactly how to divide the program into methods. This process lets you design as you go along.

7.6 Local variables

You might wonder how we can use the same variable `i` in both `printMultiples` and `printMultTable`. Didn't I say that you can only declare a variable once? And doesn't it cause problems when one of the methods changes the value of the variable?

The answer to both questions is "no," because the `i` in `printMultiples` and the `i` in `printMultTable` are *not the same variable*. They have the same name, but they do not refer to the same storage location in memory, and changing the value of one has no effect on the other.

Variables declared inside a method definition are called **local variables** because they only exist inside the method. You cannot access a local variable from outside its "home" method, and you are free to have multiple variables with the same name, as long as they are not in the same method.

Although it can be confusing, there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you

avoid using them in one method just because you used them somewhere else, you make the program harder to read.

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`:

```
1  public static void printMultTable(int high) {
2    int i = 1;
3    while (i <= high) {
4      printMultiples(i);
5      i = i + 1;
6    }
7  }
```

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.

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

```
1  public static void printMultiples(int n, int high) {
2    int i = 1;
3    while (i <= high) {
4      System.out.print(n*i + "   ");
5      i = i + 1;
6    }
7    System.out.println("");
8  }
```

```
9
10  public static void printMultTable(int high) {
11    int i = 1;
12    while (i <= high) {
13      printMultiples(i, high);
14      i = i + 1;
15    }
16  }
```

Notice that when I added a new parameter, I had to change the first line, 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 it has capabilities you did not plan. 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

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

to

```
1     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 Glossary

loop: A statement that executes repeatedly while 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.

program development: A process for writing programs. So far we have seen "incremental development" and "encapsulation and generalization".

7.9 Exercises

Exercise 7.1. Consider the following code:

```
public static void main(String[] args) {
    loop(10);
}

public static void loop(int n) {
    int i = n;
    while (i > 0) {
        System.out.println(i);
        if (i%2 == 0) {
            i = i/2;
```

```
11          } else {
12              i = i+1;
13          }
14      }
15  }
```

1. Draw a table that shows the value of the variables `i` and `n` during the execution of `loop`. The table should contain one column for each variable and one line for each iteration.

2. What is the output of this program?

Exercise 7.2. Consider the program `Collatz.java`, Program 7.1. Determine the outputs of the program for each of the following method calls. You only need to give the first 7 outputs. If the program would continue execution beyond that point, you can simply write ...

1. `sequence(7);`

2. `sequence(32);`

3. `sequence(-5);`

What can you conclude about the method call `sequence(-5);`?

Exercise 7.3. Let's say you are given a number, a, and you want to find its square root. One way to do that is to start with a very rough guess about the answer, x_0, and then improve the guess using the following formula:

$$x_1 = (x_0 + a/x_0)/2$$

For example, if we want to find the square root of 9, and we start with $x_0 = 6$, then $x_1 = (6 + 9/6)/2 = 15/4 = 3.75$, which is closer.

We can repeat the procedure, using x_1 to calculate x_2, and so on. In this case, $x_2 = 3.075$ and $x_3 = 3.00091$. So that is converging very quickly on the right answer(which is 3).

Write a method called `squareRoot` that takes a `double` as a parameter and that returns an approximation of the square root of the parameter, using this technique. You may not use `Math.sqrt`.

As your initial guess, you should use $a/2$. Your method should iterate until it gets two consecutive estimates that differ by less than 0.0001; in other words, until the absolute value of $x_n - x_{n-1}$ is less than 0.0001. You can use `Math.abs` to calculate the absolute value.

Exercise 7.4. In Exercise 6.6 we wrote a recursive version of `power`, which takes a double `x` and an integer `n` and returns x^n. Now write an iterative method to perform the same calculation.

Exercise 7.5. Section 6.3 presents a recursive method that computes the factorial function. Write an iterative version of `factorial`.

Exercise 7.6. One way to calculate e^x is to use the infinite series expansion

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + ...$$

If the loop variable is named `i`, then the ith term is $x^i/i!$.

1. Write a method called `myexp` that adds up the first `n` terms of this series. You can use the `factorial` method from Section 6.3 or your iterative version from the previous exercise.

2. You can make this method much more efficient if you realize that in each iteration the numerator of the term is the same as its predecessor multiplied by `x` and the denominator is the same as its predecessor multiplied by `i`. Use this observation to eliminate the use of `Math.pow` and `factorial`, and check that you still get the same result.

3. Write a method called `check` that takes a single parameter, `x`, and that prints the values of `x`, `Math.exp(x)` and `myexp(x)` for various values of `x`. The output should look something like:

   ```
   1.0     2.708333333333333       2.718281828459045
   ```

   HINT: you can use the String `"\t"` to print a tab character between columns of a table.

4. Vary the number of terms in the series (the second argument that `check` sends to `myexp`) and see the effect on the accuracy of the result. Adjust this value until the estimated value agrees with the "correct" answer when `x` is 1.

5. Write a loop in `main` that invokes `check` with the values 0.1, 1.0, 10.0, and 100.0. How does the accuracy of the result vary as `x` varies? Compare the number of digits of agreement rather than the difference between the actual and estimated values.

6. Add a loop in `main` that checks `myexp` with the values -0.1, -1.0, -10.0, and -100.0. Comment on the accuracy.

Exercise 7.7. One way to evaluate $\exp(-x^2)$ is to use the infinite series expansion

$$\exp(-x^2) = 1 - x^2 + x^4/2 - x^6/6 + \ldots$$

In other words, we need to add up a series of terms where the ith term is equal to $(-1)^i x^{2i}/i!$. Write a method named `gauss` that takes `x` and `n` as arguments and that returns the sum of the first `n` terms of the series. You should not use `factorial` or `pow`.

Chapter 8

Strings and things

8.1 Characters

In Java and other object-oriented languages, **objects** are collections of related data that come with a set of methods. These methods operate on the objects, performing computations and sometimes modifying the object's data.

`Strings` are objects, so you might ask "What is the data contained in a `String` object?" and "What are the methods we can invoke on `String` objects?" The components of a `String` object are letters or, more generally, characters. Not all characters are letters; some are numbers, symbols, and other things. For simplicity I call them all letters. There are many methods, but I use only a few in this book. The rest are documented at http://download.oracle.com/javase/8/docs/api/java/lang/String.html.

The first method we will look at is `charAt`, which allows you to extract letters from a `String`. `char` is the variable type that can store individual characters (as opposed to strings of them).

`char`s work just like the other types we have seen:

```
char ltr = 'c';
if (ltr == 'c') {
  System.out.println(ltr);
}
```

Character values appear in single quotes, like `'c'`. Unlike string values (which appear in double quotes), character values can contain only a single letter or symbol.

Here's how the `charAt` method is used:

```
String fruit = "banana";
char letter = fruit.charAt(1);
System.out.println(letter);
```

`fruit.charAt()` means that I am invoking the `charAt` method on the object named `fruit`. I am passing the argument `1` to this method, which means that I want 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 technical reasons, computer scientists start counting from zero. The 0th letter ("zeroeth") of `"banana"` is `b`. The 1th letter ("oneth") is `a` and the 2th ("twooth") letter is `n`.

If you want the zereoth letter of a string, you have to pass 0 as an argument:

```
char letter = fruit.charAt(0);
```

8.2 Length

The next `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 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);
```

8.3 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do some computation with 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 a member of an ordered set, in this case the string of characters. The index indicates (hence the name) which one you want.

8.4 Run-time errors

Way back in Section 1.3.2 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**.

You probably haven't seen many run-time errors, because we haven't been doing many things that can cause one. Well, now we are. If you use the `charAt` method and provide an index that is negative or greater than `length-1`, it **throws** an exception. You can think of "throwing" an exception like throwing a tantrum.

When that happens, Java prints an error message with the type of exception and a **stack trace**, which shows the methods that were running when

the exception occurred. Here is an example:

```
public class BadString {

    public static void main(String[] args) {
        processWord("banana");
    }

    public static void processWord(String s) {
        char c = getLastLetter(s);
        System.out.println(c);
    }

    public static char getLastLetter(String s) {
        int index = s.length();         // WRONG!
        char c = s.charAt(index);
        return c;
    }
}
```

Notice the error in `getLastLetter`: the index of the last character should be `s.length()-1`. Here's what you get:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 6
        at java.lang.String.charAt(String.java:694)
        at BadString.getLastLetter(BadString.java:24)
        at BadString.processWord(BadString.java:18)
        at BadString.main(BadString.java:14)
```

Then the program ends. The stack trace can be hard to read, but it contains a lot of information. Exercise 8.2 has some questions to help you understand stack traces.

8.5 Reading documentation

If you go to
http://download.oracle.com/javase/8/docs/api/java/lang/String.html.
and click on `charAt`, you get the following documentation (or something like it):

```
public char charAt(int index)
```

```
Returns the char value at the specified index. An index ranges
from 0 to length() - 1. The first char value of the sequence is
at index 0, the next at index 1, and so on, as for array indexing.

Parameters: index - the index of the char value.

Returns: the char value at the specified index of this string.
  The first char value is at index 0.

Throws: IndexOutOfBoundsException - if the index argument is
  negative or not less than the length of this string.
```

The first line is the method's **signature** (also called the method **header**), which specifies the name of the method, the type of the parameters, and the return type.

The next line describes what the method does. The following lines explain the parameters and return values. In this case the explanations are redundant, but the documentation is supposed to fit a standard format. The last line describes the exceptions this method might throw.

It might take some time to get comfortable with this kind of documentation, but it is worth the effort.

8.6 The indexOf method

`indexOf` is the inverse 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 throws an exception. `indexOf` fails if the character does not appear in the string, and returns the value `-1`.

```
1    String fruit = "banana";
2    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.

To find subsequent appearances, there is another version of `indexOf`. It takes a second argument that indicates where in the string to start looking.

For an explanation of this kind of overloading, see Section 4.12.

If we invoke:

```
int index = fruit.indexOf('a', 2);
```

it starts at the twoeth letter (the first `n`) and finds the second `a`, which is at index 3. If the letter happens to appear at the starting index, the starting index is the answer. So

```
int index = fruit.indexOf('a', 5);
```

returns 5.

8.7 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**. When we exit the loop, `count` contains the result: the total number of a's.

8.8 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 `doubles`, `booleans` or `Strings`.

Both the `++` or `--` operators can come before or after the variable. In other words, both `i++;` and `++i;` are syntactically legal statements which increase the value of `i` by 1. If the operator comes *before* the variable, we call it a **pre-increment** or **pre-decrement operator**. If the operator comes *after* the variable, we call it the **post-increment** or **post-decrement** operator.

If you use a pre- or post- decrement or increment operators as a single statement, you won't see anything unexpected happen.

```
int x = 5;
System.out.println(x);   // displays the value 5
x++;                     // could change this to ++x; too
System.out.println(x);   // displays the value 6
```

However, you can also use these operators as part of other statements. For example:

```
int i = 4;
System.out.println(i++);
```

Looking at this, it is not clear whether the increment will take effect before or after the value is printed. In other words, we're not sure which of the following two snippets of code is equivalent to the one above:

```
int i = 4;
System.out.println(i);
i = i + 1;
```

or

```
int i = 4;
i = i + 1;
System.out.println(i);
```

To help us out, just remember the meanings of "pre" (before) and "post" (after). If we use the pre- operators, then the increment or decrement happens *before* the rest of the statement. If we use the post- operators, then the increment or decrement happens *after* the statement. So the code

```
int i = 4;
System.out.println(--i);
```

will print the value 3 because the pre-decrement operator will subtract 1 from `i` before the `println` statement executes. However, the code

```
int i = 4;
System.out.println(i--);
```

will print the value 4, because the post-decrement operator will subtract 1 from `i` after the `println` statement executes.

Because complex statements like this tend to be confusing, I discourage you from using them. However, you may encounter them in other people's code and it is important to understand how they work.

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.

8.9 Shortcut Operators

Along with our increment and decrement operators in the last section, we'll now introduce some more operators which shorten frequently typed mathematical expressions. For example, you have probably often typed statements like:

```
index = index + 1;
val = val - 15;
x = x * 2;
```

All of these statements follow a common pattern of altering a variable's value by doing some elementary math (adding to it, subtracting from it, etc) and then assigning the resulting value back to the variable.

Instead of typing the above statements, we can use **shortcut operators** to shorten these statements:

```
index += 1;
```

```
    val -= 15;
    x *= 2;
```

These shortcut operators combine the mathematical operator with the assignment operator. Each of these shortcut operators is equivalent in functionality to the longer way of separately using the mathematical and assignment operators. We can also use `/=`, or `%=` for the division and modulo operations.

8.10 Character arithmetic

It may seem odd, but you can do arithmetic with characters. 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.

Memorizing an ASCII table is beyond the scope of this class (fortunately!), but you can take a look at one found here: http://www.asciitable.com/.

You'll see that all uppercase letters are sequential, as are all lowercase letters and digits. Various symbols are scattered throughout the table. There are also "unprintable" characters occurring mostly at low ASCII values which

are leftovers from days when computer input came from typewriter like interfaces rather than our modern hardware.

Since the letters are sequential, 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:

```
1     char letter = 'J';
2     while (letter <= 'Q') {
3       System.out.println(letter + "ack");
4       letter++;
5     }
6
```

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.

8.11 `Strings` are immutable

As you read 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"`.

8.12 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 (we'll learn why not later on).

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 `String`s, 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 weird, too. 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. Take a look

at an ASCII table and see if you can figure out how we could easily arrive at the value of 8 for the letters 'd' and 'l'.

Just for completeness, I should admit that it is *legal*, but very seldom *correct*, to use the `==` operator with `String`s. I explain why in Section 12.5; for now, don't do it.

8.13 Glossary

counter: A variable used to count something, usually initialized to zero and then incremented.

exception: A run-time error.

header: The first line of a method, which specifies the name, parameters and return type.

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

object: A collection of related data that comes with a set of methods that operate on it. The objects we have used so far are `String`s, Bugs, Rocks, and the other GridWorld objects.

pre-increment: Increase the value of a variable by one before any other actions are taken. The increment operator in Java is `++`.

pre-decrement: Decrease the value of a variable by one before any other actions are taken. The decrement operator in Java is `--`.

post-increment: Increase the value of a variable by one after any other actions are taken. The increment operator in Java is `++`.

post-decrement: Decrease the value of a variable by one after any other actions are taken. The decrement operator in Java is `--`.

shortcut operator : Operator which allows us to express both a mathematical operation and an assignment operator more concisely.

stack trace: A report that shows the state of a program when an exception occurs.

throw: Cause an exception.

traverse: To iterate through all the elements of a set performing a similar operation on each.

8.14 Exercises

Exercise 8.1. Write a method that takes a `String` as an argument and that prints the letters backwards all on one line.

Exercise 8.2. Read the stack trace in Section 8.4 and answer these questions:

- What kind of Exception occurred, and what package is it defined in?
- What is the value of the index that caused the exception?
- What method threw the exception, and where is that method defined?
- What method invoked `charAt`?
- In `BadString.java`, what is the line number where `charAt` was invoked?

Exercise 8.3. Encapsulate the code in Section 8.7 in a method named `countLetters`, and generalize it so that it accepts the string and the letter as arguments.

Then rewrite the method so that it uses `indexOf` to locate the a's, rather than checking the characters one by one.

Exercise 8.4. The purpose of this exercise is to review encapsulation and generalization.

1. Encapsulate the following code fragment, transforming it into a method that takes a String as an argument and that returns the final value of `count`.

2. In a sentence or two, describe what the resulting method does (without getting into the details of how).

3. Now that you have generalized the code so that it works on any String, what could you do to generalize it more?

```
String s = "((3 + 7) * 2)";
int len = s.length();

int i = 0;
int count = 0;

while (i < len) {
    char c = s.charAt(i);

    if (c == '(') {
       count = count + 1;
    } else if (c == ')') {
       count = count - 1;
    }
    i = i + 1;
}

System.out.println(count);
```

Exercise 8.5. The point of this exercise is to explore Java types and fill in some of the details that aren't covered in the chapter.

1. Create a new program named `Test.java` and write a `main` method that contains expressions that combine various types using the `+` operator. For example, what happens when you "add" a `String` and a `char`? Does it perform addition or concatenation? What is the type of the result? (How can you determine the type of the result?)

2. Make a bigger copy of the following table and fill it in. At the intersection of each pair of types, you should indicate whether it is legal to use the `+` operator with these types, what operation is performed (addition or concatenation), and what the type of the result is.

	boolean	char	int	String
boolean				
char				
int				
String				

3. Think about some of the choices the designers of Java made when they filled in this table. How many of the entries seem unavoidable, as if there were no other choice? How many seem like arbitrary choices from several equally reasonable possibilities? How many seem problematic?

4. Here's a puzzler: normally, the statement `x++` is exactly equivalent to `x = x + 1`. But if `x` is a `char`, it's not! 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.

Exercise 8.6. What is the output of this program? Describe in a sentence what `mystery` does (not how it works).

```
public class Mystery {

    public static String mystery(String s) {
        int i = s.length() - 1;
        String total = "";

        while (i >= 0 ) {
            char ch = s.charAt(i);
            System.out.println(i + "      " + ch);

            total = total + ch;
            i--;
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(mystery("Allen"));
    }
}
```

Exercise 8.7. A friend of yours shows you the following method and explains that if `number` is any two-digit number, the program will output the number backwards. He claims that if `number` is 17, the method will output `71`.

Is he right? If not, explain what the program actually does and modify it so that it does the right thing.

```
    int number = 17;
    int lastDigit = number%10;
    int firstDigit = number/10;
    System.out.println(lastDigit + firstDigit);
```

Exercise 8.8. What is the output of the following program?

```
public class Enigma {

    public static void enigma(int x) {
        if (x == 0) {
            return;
        } else {
            enigma(x/2);
        }

        System.out.print(x%2);
    }

    public static void main(String[] args) {
        enigma(5);
        System.out.println("");
    }
}
```

Explain in 4-5 words what the method `enigma` really does.

Exercise 8.9. 1. Create a new program named `Palindrome.java`.

2. Write a method named `first` that takes a String and returns the first letter, and one named `last` that returns the last letter.

3. Write a method named `middle` that takes a String and returns a substring that contains everything *except* the first and last characters.

   Hint: read the documentation of the `substring` method in the `String` class. Run a few tests to make sure you understand how `substring` works before you try to write `middle`.

   What happens if you invoke `middle` on a string that has only two letters? One letter? No letters?

4. The usual definition of a palindrome is a word that reads the same both forward and backward, like "otto" and "palindromeemordnilap." An alternative way to define a property like this is to specify a way of testing for the property. For example, we might say, "a single letter is a palindrome, and a two-letter word is a palindrome if the letters are the same, and any other word is a palindrome if the first letter is the same as the last and the middle is a palindrome."

Write a recursive method named `isPalindrome` that takes a `String` and that returns a boolean indicating whether the word is a palindrome or not.

5. Once you have a working palindrome checker, look for ways to simplify it by reducing the number of conditions you check. Hint: it might be useful to adopt the definition that the empty string is a palindrome.

6. On a piece of paper, figure out a strategy for checking palindromes iteratively. There are several possible approaches, so make sure you have a solid plan before you start writing code.

7. Implement your strategy in a method called `isPalindromeIter`.

8. Optional: Appendix B provides code for reading a list of words from a file. Read a list of words and print the palindromes.

Exercise 8.10. A word is said to be "abecedarian" if the letters in the word appear in alphabetical order. For example, the following are all 6-letter English abecedarian words.

abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

1. Describe a process for checking whether a given word (String) is abecedarian, assuming that the word contains only lower-case letters. Your process can be iterative or recursive.

2. Implement your process in a method called `isAbecedarian`.

Exercise 8.11. A dupledrome is a word that contains only double letters, like "llaammaa" or "ssaabb". I conjecture that there are no dupledromes in common English use. To test that conjecture, I would like a program that reads words from the dictionary one at a time and checks them for dupledromity.

Write a method called `isDupledrome` that takes a String and returns a boolean indicating whether the word is a dupledrome.

Exercise 8.12. 1. The Captain Crunch decoder ring works by taking each letter in a string and adding 13 to it. For example, 'a' becomes 'n' and 'b' becomes 'o'. The letters "wrap around" at the end, so 'z' becomes 'm'.

Write a method that takes a String and that returns a new String containing the encoded version. You should assume that the String contains upper and lower case letters, and spaces, but no other punctuation. Lower case letters should be tranformed into other lower case letters; upper into upper. You should not encode the spaces.

2. Generalize the Captain Crunch method so that instead of adding 13 to the letters, it adds any given amount. Now you should be able to encode things by adding 13 and decode them by adding -13. Try it.

Chapter 9

Arrays

An **array** is a set of values where each value 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, array 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`. We'll talk more about the special value `null` in Section 10.9, but for now, just remember that you can't use or manipulate an array until you've allocated space in memory for a specific number of values in an array. To do this and create the array itself, use `new`.

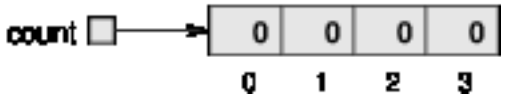
```
count = new int[4];
values = new double[size];
```

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 the value of the variable `size`. You can use any integer expression as an array size.

When you declare an array variable, you get a reference to an array. In other words, the value that is assigned to the array variable is the address of the array in memory. Technically, the address is of the first value in the array, and Java also stores the number of total values in your array. This is

fundamentally different than variables we have seen in past. Sometimes, we refer to these types of variables as **reference variables** to emphasize that the value stored to them is a reference to a memory location. We will talk about this again when we cover objects in Chapters 10 and 11

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 an array of `int`s, the elements are initialized to zero. If you allocate an array of floating point numbers, the elements are initialized to `0.0`. If you create an array of `boolean`s, the initial value of all elements is `false`.

9.1 Arrays with initial values

There is another, less commonly used way to create an array. In this case, we will create an array and assign it initial values in one statement. The following two statements create an array of integers and an array of doubles, respectively.

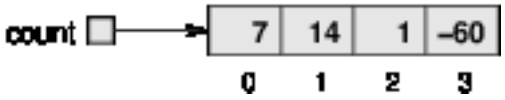
```
1    int[] count = {3, 2, -18};
2    double[] values = {1.4, -5.2, 0.0};
```

9.2 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:

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

These are all legal assignment statements. Here is the result of this code fragment:



The elements of the array are numbered from 0 to 3, which means that there is no element with the index 4. This should sound familiar, since we saw the same thing with `String` indices. Nevertheless, it is a common error to go beyond the bounds of an array, which throws an `ArrayOutOfBoundsException`.

9.3 Printing values in an array

When students wish to inspect the values in an array, they often try something like:

```
1  int[] values = {2, 5, 6};
2  System.out.println( values );
```

However, instead of seeing something like `[2, 5, 6]` or another similar representation of the array, you'll see something like:

```
[I@48e61e
```

The actual value you see printed will vary, but the general format will be the same. In fact, this emphasizes the fact we talked about earlier: that array variables are reference variables. What's being printed is the actual value assigned to the array variable which is a *reference* to the array in memory (ie a memory address).

If we want to print the values in an array, we will need to use a loop to iterate over the elements in an array and print their values.

We will need to write a standard `while` loop that counts from 0 up to (and including) the last element in the array, and when the loop variable `i` is more than the last index in the array, the condition will fail and the loop will terminate. Thus, for the example above, we want the body of the loop to execute when `i` is 0, 1, 2 or 3.

Each time through the loop we will then use `i` as an index into the array, printing the `i`th element. This type of array inspection is very common, and we refer to it as **traversing** an array. You can use any expression as an index into an array, as long as it has type `int`. One of the most common ways of traversing an array is to index an array is with a loop variable.

For example:

```
1    int i = 0;
```

```
    while (i < 4) {
        System.out.print( count[i] + " ");
        i++;
    }
```

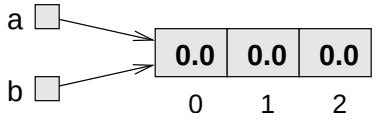
This code will print out the elements in our array, separated by a space.

9.4 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 `double`s, and sets two different variables to refer to it. This situation is a form of aliasing which we'll cover more in Section 10.8.



Any changes in either array will be reflected in the other. This is not usually the behavior you want and can be very confusing; more often you want to make *independent* copies. To do this, you need to allocate a new array and copy elements from one to the other.

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

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

9.5 `for` loops

The loops we have written 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 another loop statement, called `for`, that expresses it more concisely. The general syntax looks like this:

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

This statement is 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++;
}
```

Just like while loops, the condition is checked before the first execution of the body of the loop. Therefore, it is possible to write for (or while) loops which never execute if the condition is immediately `false`.

For loops and while loops are logically equivalent. That is, anything you write with a while loop, you can express with a for loop and vice versa. Which loop you chose depends on your personal coding style and the problem you are trying to solve. Some problems lend themselves more easily to one type of loop or the other.

You might note that in the previous example, we declare and initialize the variable `i` inside the for loop. This effectively makes the scope of `i` only the for loop, and you will not be able to access this variable outside of the loop.

It is also worth noting that you can "count down" with for loops as well. For example

```
for(int i = 10; i <= 10; i--) {
    System.out.println(i);
}
System.out.println("BLASTOFF!");
```

will print out a countdown to blastoff.

Technically, you can also change the value of your loop variable inside the loop. However, this is needlessly confusing and violates the spirit of a for loop. Consider this code:

```
for(int i = 10; i <= 10; i--) {
    System.out.println(i);
    i += 2;
}
System.out.println("BLASTOFF!");
```

Trace this code and determine what happens.

9.6 Array length

All arrays have one variable associated with them which stores the length of the array. This variable is named `length`. 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 throw an exception. This code assumes that the array `b` contains at least as many elements as `a`.

We've seen this general idea before when working with `String`s and the length of `String`s. However, there's a crucial different. For arrays, `length` is a variable and for `String`s, `length()` is a method. Note that there are `()` only when working with `String`s.

9.7 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. But for some applications we want the computer to be unpredictable. Games are an obvious example, but there are 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 method that generates **pseudorandom** numbers, which may not be truly random, 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. To be precise, it is greater than or equal to 0.0 and strictly less than 1.0. Each time you invoke `random` you get the next number in a pseudorandom sequence. To see a sample, run this loop:

```
1    for (int i = 0; i < 10; i++) {
2        double x = Math.random();
3        System.out.println(x);
4    }
```

To generate a random `double` between 0.0 and an upper bound like `high`, you can multiply `x` by `high`.

9.8 Array of random numbers

How would you generate a random integer between `low` and `high`? If your implementation of `randomInt` is correct, then every value in the range from `low` to `high-1` should have the same probability. If you generate a long series of numbers, every value should appear, at least approximately, the same number of times.

One way to test your method is to generate a large number of random values, store them in an array, and count the number of times each value occurs.

The following method takes a single argument, the size of the array. It allocates a new array of integers, fills it with random values, and returns a reference to the new array.

```
public static int[] randomArray(int n) {
    int[] a = new int[n];
    for (int i = 0; i<a.length; i++) {
        a[i] = randomInt(0, 100);
    }
    return a;
}
```

The return type is `int[]`, which means that this method returns an array of integers. To test this method, it is convenient to have a method that prints the contents of an array.

```
public static void printArray(int[] a) {
    for (int i = 0; i < a.length; i++) {
        System.out.println(a[i]);
    }
}
```

The following code generates an array and prints it:

```
int numValues = 8;
int[] array = randomArray(numValues);
printArray(array);
```

On my machine the output is

```
27
6
54
62
54
2
44
81
```

which is pretty random-looking. Your results will probably differ.

If these were exam scores (and they would be pretty bad exam scores) the teacher might present the results to the class in the form of a **histogram**, which is a set of counters that keeps track of the number of times each value appears.

For exam scores, we might have ten counters to keep track of how many students scored in the 90s, the 80s, etc. The next few sections develop code to generate a histogram.

9.9 Counting

A good approach to problems like this is to think of simple methods that are easy to write, then combine them into a solution. This process is called **bottom-up development**. See http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

It is not always obvious where to start, but a good approach is to look for subproblems that fit a pattern you have seen before.

In Section 8.7 we saw a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like an array or a string.
- A test that you can apply to each element in the container.
- A counter that keeps track of how many elements pass the test.

In this case, the container is an array of integers. The test is whether or not a given score falls in a given range of values.

Here is a method called `inRange` that counts the number of elements in an array that fall in a given range. The parameters are the array and two integers that specify the lower and upper bounds of the range.

```
public static int inRange(int[] a, int low, int high) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] >= low && a[i] < high) count++;
    }
    return count;
}
```

I wasn't specific about whether something equal to `low` or `high` falls in the range, but you can see from the code that `low` is in and `high` is out. That keeps us from counting any elements twice.

Now we can count the number of scores in the ranges we are interested in:

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
```

```
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

9.10 The histogram

This code is repetitious, but it is acceptable as long as the number of ranges is small. But imagine that we want to keep track of the number of times each score appears, all 100 possible values. Would you want to write this?

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
...
int count3 = inRange(scores, 99, 100);
```

I don't think so. What we really want is a way to store 100 integers, preferably so we can use an index to access each one. Hint: array.

The counting pattern is the same whether we use a single counter or an array of counters. In this case, we initialize the array outside the loop; then, inside the loop, we invoke `inRange` and store the result:

```
    int[] counts = new int[100];

    for (int i = 0; i < counts.length; i++) {
        counts[i] = inRange(scores, i, i+1);
    }
```

The only tricky thing here is that we are using the loop variable in two roles: as in index into the array, and as the parameter to `inRange`.

9.11 A single-pass solution

This code works, but it is not as efficient as it could be. Every time it invokes `inRange`, it traverses the entire array. As the number of ranges increases, that gets to be a lot of traversals.

It would be better to make a single pass through the array, and for each value, compute which range it falls in. Then we could increment the appropriate counter. In this example, that computation is trivial, because we can use the value itself as an index into the array of counters.

Here is code that traverses an array of scores once and generates a histogram.

```
    int[] counts = new int[100];

    for (int i = 0; i < scores.length; i++) {
        int index = scores[i];
        counts[index]++;
    }
```

9.12 Passing arrays to methods

In past chapters, we learned about the concept of disjoint scope. This means that changes made to the value of a parameter variable inside a method do not affect the variable originally used as an argument to the method. For example, consider the program below:

```
public class ArrayParameters {
  public static void printArray(int[] n) {
    System.out.print("[");
    for(int i = 0; i < n.length-1; i++) {
      System.out.print(n[i] + " ");
    }
    System.out.println(n[n.length-1] + "]");
  }

  public static void intParam(int n) {
    n = n + 6;
  }

  public static void arrayParam1(int[] n) {
    n = new int[4];
    n[0] = 4;
  }

  public static void arrayParam2(int[] n) {
    n[0] = 4;
  }

  public static void main(String[] args) {
    int n = 4;
    System.out.println(n);
    intParam(n);
```

```
27      System.out.println(n);
28
29      int[] x = {1, 2, 3, 4};
30      printArray(x);
31      arrayParam1(x);
32      printArray(x);
33
34      arrayParam2(x);
35      printArray(x);
36    }
37  }
```

Program 9.1: ArrayParameters.java

There are four methods in this program: `printArray, intParam, arrayParam1, arrayParam2,` and `main`. `printArray` is just a helper method to let us easily print the values in our array in a nice format.

If we run the program, we get the output:

```
4
4
[1 2 3 4]
[1 2 3 4]
[4 2 3 4]
```

The first two lines of output just demonstrate what we already know. First, the `main` method first establishes a variable `n`, prints the value of the variable, invokes the method `intParam` with it, and then prints the value again. We already know that even though the parameter variable in the method `intParam` is named the same thing, it is a different variable and changing its value does not effect the value of the variable `n` in `main`.

However, let's look at the next two lines of output, which display the values in the array `x` from the calls to `printArray` on lines 30 and 32. On line 29, we declare and initialize an array and assign the reference to the variable `x`. We print the array and then invoke `arrayParam1` with the array variable. As expected, the values in the array `x` are the same before and after the method call.

But take a look at what happens after that! We invoke `arrayParam2` and the values in the array are changed after the method completes. What happened?!

Java is a **pass-by-value** language. This means that if we use a variable as an argument to a method, the value assigned to the variable is copied to the

parameter variable. Then, any changes made to the value of the parameter variable do not effect the original variable.

Take a look at line 15 in the `arrayParam1` method. This line assigns a new value to the parameter variable `n`. After this method ends, this parameter variable is no longer in scope. Since we know Java is a pass-by-value language, we can see that assigning a new reference to the parameter variable `n` will not affect the reference assigned to the local variable `x` in `main`.

If you look at the `arrayParam2` method, you will see there is not any code which makes a new array reference and assigns it to the parameter variable. In other words, the value of `x` (which is a reference to a memory address) is assigned to the parameter variable `n`. We then tell Java, "Hey, go to the memory location specified by `n` and alter the first integer in the array to be 0." Since reference variables are really memory addresses, we can alter arrays inside methods!

9.13 Glossary

array: A collection of values, where all the values have the same type, and each value is identified by an index.

deterministic: A program that does the same thing every time it is invoked.

element: One of the values in an array. The `[]` operator selects elements.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

index: An integer variable or value used to indicate an element of an array.

: A parameter passing mechanism in which the value of a variable used as an argument to a method is copied to the parameter variable of the method.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

reference: A value that refers to the location of an object. In a state diagram, a reference appears as an arrow.

reference variable: A variable whose value is a reference rather then a primitive value.

traverse: To iterate over an array and examine each element

9.14 Exercises

Exercise 9.1. 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.

Exercise 9.2. Write a method called `randomDouble` that takes two doubles, `low` and `high`, and that returns a random double x so that $low \leq x < high$.

Exercise 9.3. Write a method called `randomInt` that takes two arguments, `low` and `high`, and that returns a random integer between `low` and `high`, not including `high`.

Exercise 9.4. Encapsulate the code in Section 9.11 in a method called `makeHist` that takes an array of scores and returns a histogram of the values in the array.

Exercise 9.5. Write a method named `areFactors` that takes an integer `n` and an array of integers, and that returns `true` if the numbers in the array are all factors of `n` (which is to say that `n` is divisible by all of them). HINT: See Exercise 5.5.

Exercise 9.6. Write a method that takes an array of integers and an integer named `target` as arguments, and that returns the first index where `target` appears in the array, if it does, and -1 otherwise.

Exercise 9.7. Some programmers disagree with the general rule that variables and methods should be given meaningful names. Instead, they think variables and methods should be named after fruit.

For each of the following methods, write one sentence that describes abstractly what the method does. For each variable, identify the role it plays.

```
1 public static int banana(int[] a) {
2     int grape = 0;
3     int i = 0;
```

```
    while (i < a.length) {
        grape = grape + a[i];
        i++;
    }
    return grape;
}

public static int apple(int[] a, int p) {
    int i = 0;
    int pear = 0;
    while (i < a.length) {
        if (a[i] == p) pear++;
        i++;
    }
    return pear;
}

public static int grapefruit(int[] a, int p) {
    for (int i = 0; i<a.length; i++) {
        if (a[i] == p) return i;
    }
    return -1;
}
```

The purpose of this exercise is to practice reading code and recognizing the computation patterns we have seen.

Exercise 9.8. 1. What is the output of the following program?

2. Draw a stack diagram that shows the state of the program just before `mus` returns.

3. Describe in a few words what `mus` does.

```
public static int[] make(int n) {
    int[] a = new int[n];

    for (int i = 0; i < n; i++) {
        a[i] = i+1;
    }
    return a;
}

public static void dub(int[] jub) {
    for (int i = 0; i < jub.length; i++) {
        jub[i] *= 2;
```

```
    }
}

public static int mus(int[] zoo) {
    int fus = 0;
    for (int i = 0; i < zoo.length; i++) {
        fus = fus + zoo[i];
    }
    return fus;
}

public static void main(String[] args) {
    int[] bob = make(5);
    dub(bob);

    System.out.println(mus(bob));
}
```

Exercise 9.9. Many of the patterns we have seen for traversing arrays can also be written recursively. It is not common to do so, but it is a useful exercise.

1. Write a method called `maxInRange` that takes an array of integers and a range of indices (`lowIndex` and `highIndex`), and that finds the maximum value in the array, considering only the elements between `lowIndex` and `highIndex`, including both ends.

   This method should be recursive. If the length of the range is 1, that is, if `lowIndex == highIndex`, we know immediately that the sole element in the range must be the maximum. So that's the base case.

   If there is more than one element in the range, we can break the array into two pieces, find the maximum in each of the pieces, and then find the maximum of the maxima.

2. Methods like `maxInRange` can be awkward to use. To find the largest element in an array, we have to provide a range that includes the entire array.

   ```
   double max = maxInRange(array, 0, a.length-1);
   ```

   Write a method called `max` that takes an array as a parameter and that uses `maxInRange` to find and return the largest value. Methods like

`max` are sometimes called **wrapper methods** because they provide a layer of abstraction around an awkward method and make it easier to use. The method that actually performs the computation is called the **helper method**.

3. Write a recursive version of `find` using the wrapper-helper pattern. `find` should take an array of integers and a target integer. It should return the index of the first location where the target integer appears in the array, or -1 if it does not appear.

Exercise 9.10. One not-very-efficient way to sort the elements of an array is to find the largest element and swap it with the first element, then find the second-largest element and swap it with the second, and so on. This method is called a **selection sort** (see http://en.wikipedia.org/wiki/Selection_sort).

1. Write a method called `indexOfMaxInRange` that takes an array of integers, finds the largest element in the given range, and returns its *index*. You can modify your recursive version of `maxInRange` or you can write an iterative version from scratch.

2. Write a method called `swapElement` that takes an array of integers and two indices, and that swaps the elements at the given indices.

3. Write a method called `selectionSort` that takes an array of integers and that uses `indexOfMaxInRange` and `swapElement` to sort the array from largest to smallest.

Exercise 9.11. Write a method called `letterHist` that takes a String as a parameter and that returns a histogram of the letters in the String. The zeroeth element of the histogram should contain the number of a's in the String (upper and lower case); the 25th element should contain the number of z's. Your solution should only traverse the String once.

Exercise 9.12. A word is said to be a "doubloon" if every letter that appears in the word appears exactly twice. For example, the following are all the doubloons I found in my dictionary.

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Write a method called `isDoubloon` that returns `true` if the given word is a doubloon and `false` otherwise.

Exercise 9.13. Two words are anagrams if they contain the same letters (and the same number of each letter). For example, "stop" is an anagram of "pots" and "allen downey" is an anagram of "well annoyed."

Write a method that takes two Strings and returns `true` if the Strings are anagrams of each other.

Optional challenge: read the letters of the Strings only once.

Exercise 9.14. In Scrabble each player has a set of tiles with letters on them, and the object of the game is to use those letters to spell words. The scoring system is complicated, but longer words are usually worth more than shorter words.

Imagine you are given your set of tiles as a String, like `"quijibo"` and you are given another String to test, like `"jib"`. Write a method called `canSpell` that takes two Strings and returns `true` if the set of tiles can be used to spell the word. You might have more than one tile with the same letter, but you can only use each tile once.

Optional challenge: read the letters of the Strings only once.

Exercise 9.15. In real Scrabble, there are some blank tiles that can be used as wild cards; that is, a blank tile can be used to represent any letter.

Think of an algorithm for `canSpell` that deals with wild cards. Don't get bogged down in details of implementation like how to represent wild cards. Just describe the algorithm, using English, pseudocode, or Java.

Chapter 10

Mutable objects

`String`s are objects, but they are atypical objects because

- They are immutable.
- They have no attributes.
- You don't have to use `new` to create one.

In this chapter, we use two objects from Java libraries, `Point` and `Rectangle`. But first, I want to make it clear that these points and rectangles are not graphical objects that appear on the screen. They are values that contain data, just like `int`s and `double`s. Like other values, they are used internally to perform computations.

In contrast, we call `int`s, `double`s, `boolean`s, and `char`s **primitive data types**. In Java, if you encounter a data type which begins with a capital letter such as `String` or `Point`, it's an object. If you encounter a data type which begins with a lowercase letter like `int` or `char`, it's a primitive type. Primitive data types hold only a single value, and there are no methods associated with a given value of that type.

10.1 Packages

The Java libraries are divided into **packages**, including `java.lang`, which contains most of the classes we have used so far, and `java.awt`, the **Abstract Window Toolkit** (AWT), which contains classes for windows, buttons, graphics, etc.

To use a class defined in another package, you have to **import** it. `Point` and `Rectangle` are in the `java.awt` package, so to import them like this:

```
1  import java.awt.Point;
2  import java.awt.Rectangle;
```

All `import` statements appear at the beginning of the program, outside the class definition.

The classes in `java.lang`, like `Math` and `String`, are imported automatically, which is why we haven't needed the `import` statement yet.

10.2 `Point` objects

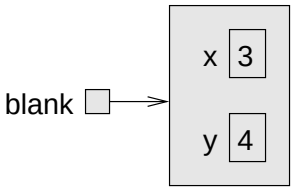
A point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up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

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

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

Line 1 is a conventional variable declaration: `blank` has type `Point`. Line 2 invokes `new`, specifies the type of the new object, and provides arguments. The arguments are the coordinates of the new point, $(3, 4)$.

The result of `new` is a **reference** to the new point, so `blank` contains a reference to the newly-created object. There is a standard way to diagram this assignment, shown in the figure.



As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a reference, which is shown graphically with an arrow. The arrow points to the object we're referring to.

The big box shows the newly-created object with the two values in it. The names `x` and `y` are the names of the **instance variables**.

Taken together, all the variables, values, and objects in a program are called the **state**. Diagrams like this that show the state of the program are called **state diagrams**. As the program runs, the state changes, so you should think of a state diagram as a snapshot of a particular point in the execution.

10.3 Instance variables

The pieces of data that make up an object are called instance variables because each object, which is an **instance** of its type, has its own copy of the instance variables.

It's like the glove compartment of a car. Each car is an instance of the type "car," and each car has its own glove compartment. If you ask me to get something from the glove compartment of your car, you have to tell me which car is yours.

Similarly, if you want to read a value from an instance variable, you have to specify the object you want to get it from. In Java this is done using "dot notation."

```
1    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`. 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.

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

Line 1 prints `(3, 4)`; Line 2 calculates the value 25.

10.4 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 + ")");
}
```

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

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

This is a standard format Java uses for printing objects. It prints the name of the type, followed by the names and values of the instance variables.

As a second example, we can rewrite the `distance` method from Section 4.10 so that it takes two `Points` as parameters instead of four `doubles`.

```
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 as Java will do the safe conversion of `int` to `double` for us; I added them as a reminder that the instance variables in a `Point` are integers.

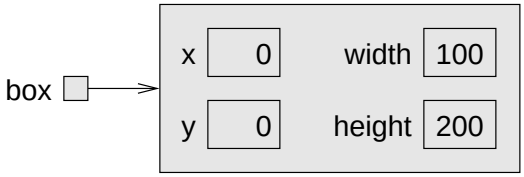
10.5 Rectangles

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

This example creates a `Rectangle` object and makes `box` refer to it.

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

This 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 Java method that knows how to print `Rectangle` objects.

10.6 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);
}
```

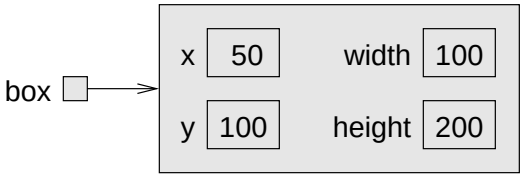
Notice that you can use `new` to create a new object, and then immediately use the result as the return value.

10.7 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 can modify the `x` and `y` values:

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

The result is shown in the figure:



We can encapsulate this code 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 do or do not modify their arguments. Later, I discuss some pros and cons of this programming style.

Java provides methods that operate on Points and Rectangles. You can read the documentation at
http://download.oracle.com/javase/8/docs/api/java/awt/Point.html
and
http://download.oracle.com/javase/8/docs/api/java/awt/Rectangle.html.

For example, `translate` has the same effect as `moveRect`, but instead of passing the Rectangle as an argument, you use dot notation:

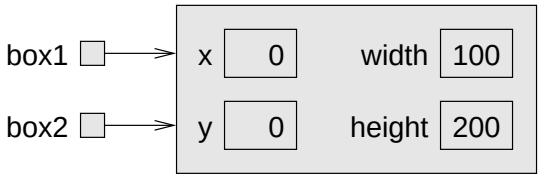
```
box.translate(50, 100);
```

10.8 Aliasing

Remember that when you assign an object to a 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:

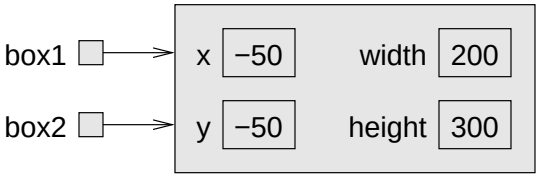


box1 and box2 refer to the same object. In other words, this object has two names, box1 and box2. When a person uses two names, it's called **aliasing**. Same thing with objects.

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:



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 can be difficult to debug. In general, aliasing should be avoided or used with care.

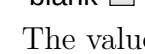
10.9 null

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 (and a Java keyword) that means "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 small square with no arrow.

If you try to use a null object, either by accessing an instance variable or

invoking a method, Java throws a `NullPointerException`, prints an error message and terminates the program. This is another example of a run-time error because it will only occur when you run the program.

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

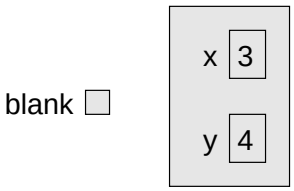
On the other hand, it is legal to pass a null object as an argument or receive one as a return value. In fact, it is common to do so, for example to represent an empty set or indicate an error condition.

10.10 Garbage collection

In Section 10.8 we 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 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 for other values.

You don't have to do anything to make garbage collection happen, and in general you will not be aware of it. But you should know that it periodically runs in the background. Other languages require you, the programmer, to handle garbage collection. However, Java is nice and handles it for you.

10.11 Objects and primitives

Back to primitive types vs. object types. Let's examine some of the differences between the two categories of types:

- Like we already discussed, primitive types begin with a lowercase letter while object types begin with an uppercase letter.
- 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. To get space for the object itself, you have to use the `new` operator.
- 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 `false` 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. 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 Java (unless you get yourself on the standards committee), but you can create new object types! We'll see how in the next chapter.

10.12 Objects and arrays

In many ways, objects behave like arrays:

- When you declare an object variable, you get a reference to an object.
- You have to use `new` to create the object itself.
- When you pass an object as an argument, you pass a reference, which means that the invoked method can change the contents of the object.

Some of the objects we looked at, like `Rectangles`, are similar to arrays in the sense that they are collections of values. This raises the question, "How is a Rectangle object different from an array of 4 integers?"

If you go back to the definition of "array" at the beginning of Chapter 9, you see one difference: the elements of an array are identified by indices, and the elements of an object have names.

Another difference is that the elements of an array have to be the same type. Objects can have instance variables with different types.

10.13 Glossary

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

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

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

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.

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

primitive type: Most basic data type in Java.

state: A complete description of all the variables and objects and their values, at a given point during the execution of a program.

state diagram: A snapshot of the state of a program, shown graphically.

10.14 Exercises

Exercise 10.1. 1. For the following program, draw a stack diagram showing the local variables and parameters of `main` and `riddle`, and show any objects those variables refer to.

2. What is the output of this program?

```
public static void main(String[] args)
{
    int x = 5;
    Point blank = new Point(1, 2);

    System.out.println(riddle(x, blank));
    System.out.println(x);
    System.out.println(blank.x);
    System.out.println(blank.y);
}

public static int riddle(int x, Point p)
{
    x = x + 7;
    return x + p.x + p.y;
}
```

The point of this exercise is to make sure you understand the mechanism for passing Objects as parameters.

Exercise 10.2. 1. For the following program, draw a stack diagram showing the state of the program just before `distance` returns. Include all variables and parameters and the objects those variables refer to.

2. What is the output of this program?

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

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);
}

public static void main(String[] args) {
    Point blank = new Point(5, 8);

    Rectangle rect = new Rectangle(0, 2, 4, 4);
    Point center = findCenter(rect);

```

```
    double dist = distance(center, blank);

    System.out.println(dist);
}
```

Exercise 10.3. The method `grow` is part of the `Rectangle` class. Read the documentation at http://download.oracle.com/javase/8/docs/api/java/awt/Rectangle

1. What is the output of the following program?
2. Draw a state diagram that shows the state of the program just before the end of `main`. Include all local variables and the objects they refer to.
3. At the end of `main`, are `p1` and `p2` aliased? Why or why not?

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

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);
}

public static void main(String[] args) {

    Rectangle box1 = new Rectangle(2, 4, 7, 9);
    Point p1 = findCenter(box1);
    printPoint(p1);

    box1.grow(1, 1);
    Point p2 = findCenter(box1);
    printPoint(p2);
}
```

Exercise 10.4. You might be sick of the factorial method by now, but we're going to do one more version.

1. Create a new program called `Big.java` and write an iterative version of `factorial`.

2. Print a table of the integers from 0 to 30 along with their factorials. At some point around 15, you will probably see that the answers are not right any more. Why not?

3. BigIntegers are Java objects that can represent arbitrarily big integers. There is no upper bound except the limitations of memory size and processing speed. Read the documentation of BigIntegers at http://download.oracle.com/javase/8/docs/api/java/math/BigInteger.html.

4. To use BigIntegers, you have to add `import java.math.BigInteger` to the beginning of your program.

5. There are several ways to create a BigInteger, but the one I recommend uses `valueOf`. The following code converts an integer to a `BigInteger`:

```
int x = 17;
BigInteger big = BigInteger.valueOf(x);
```

   Type in this code and try it out. Try printing a BigInteger.

6. Because BigIntegers are not primitive types, the usual math operators don't work. Instead we have to use methods like `add`. To add two BigIntegers, invoke `add` on one and pass the other as an argument. For example:

```
BigInteger small = BigInteger.valueOf(17);
BigInteger big = BigInteger.valueOf(1700000000);
BigInteger total = small.add(big);
```

   Try out some of the other methods, like `multiply` and `pow`.

7. Convert `factorial` so that it performs its calculation using BigIntegers and returns a BigInteger as a result. You can leave the parameter alone—it will still be an integer.

8. Try printing the table again with your modified factorial method. Is it correct up to 30? How high can you make it go? I calculated the factorial of all the numbers from 0 to 999, but my machine is pretty slow, so it took a while. The last number, 999!, has 2565 digits.

Exercise 10.5. Many encryption techniques depend on the ability to raise large integers to an integer power. Here is a method that implements a (reasonably) fast technique for integer exponentiation:

```
public static int pow(int x, int n) {
    if (n == 0) return 1;

    // find x to the n/2 recursively
    int t = pow(x, n/2);

    // if n is even, the result is t squared
    // if n is odd, the result is t squared times x

    if (n%2 == 0) {
      return t*t;
    } else {
      return t*t*x;
    }
}
```

The problem with this method is that it only works if the result is smaller than 2 billion. Rewrite it so that the result is a `BigInteger`. The parameters should still be integers, though.

You can use the BigInteger methods `add` and `multiply`, but don't use `pow`, which would spoil the fun.

Chapter 11

Create your own objects

11.1 Class definitions and object types

Way back in Section 1.5 when we defined the class `Hello`, we also created an object type named `Hello`. We didn't create any variables with type `Hello`, and we didn't use `new` to create any `Hello` objects, but we could have!

That example doesn't make much sense, since there is no reason to create a `Hello` object, and it wouldn't do much if we did. In this chapter, we will look at class definitions that create *useful* object types.

Here are the most important ideas in this chapter:

- 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; that is, it is an instance of some class.
- When you invoke `new` 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.
- The methods that operate on a type are defined in the class definition for that type.

Here are some syntax issues about class definitions:

- Class names (and hence object types) should 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. Other classes *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 class, `Time`.

11.2 Time

A common motivation for creating an object type is to encapsulate related data in an object that can be treated as a single unit. We have already seen two types like this, `Point` and `Rectangle`.

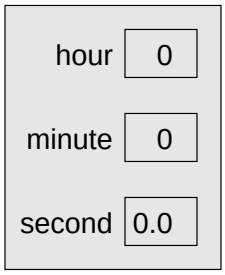
Another example, which we will implement ourselves, is `Time`, which represents the time of day. The data encapsulated in a Time object are an hour, a minute, and a number of seconds. Because every `Time` object contains these data, we need instance variables to hold them.

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`.

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;
}
```

By itself, this code fragment is a legal class definition. The state diagram for a `Time` object looks like this:



After declaring the instance variables, the next step is to define a constructor for the new class.

11.3 Constructors

Constructors initialize instance variables when we use `new` to create an instance of an object. The syntax for constructors is similar to that of other methods, with three exceptions that make constructors easy to recognize:

- 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:

```
1    public Time() {
2        this.hour = 0;
3        this.minute = 0;
4        this.second = 0.0;
5    }
```

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. Each line of the constructor initializes an instance variable to a default value (in this case, midnight). The name `this` is a special keyword that refers to 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 can't make an assignment to it. `this` is created by the system; all you have to do is initialize its instance variables.

A common error when writing constructors is to put a `return` statement at the end. Resist the temptation.

11.4 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 `new` with the parameters of the constructors.

It is common to have one constructor that takes no arguments (shown above but repeated here as well), and one constructor that takes a parameter list identical to the list of instance variables. For example:

```
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;
}
```

The names and types of the parameters are 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 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.

The last type of constructor you will often see is what's known as a **copy constructor**. This constructor takes an object of the same type and copies the instance variables of that object to the new object we are creating. Many students do something like this when they want to copy an object:

```
Time x = new Time(6, 30, 1.5);
```

```
Time y = x;
```

But in Section 10.8 we saw that this doesn't make an *independent* copy. If we change `x`, we change `y` and vice versa. We've created an alias rather than a copy that we can change independently. We will often define a copy constructor to allow us to easily make independent copies. Here is a copy constructor for the `Time` class:

```
public Time(Time t) {
  this.hour = t.hour;
  this.minute = t.minute;
  this.second = t.second;
}
```

This constructor receives an argument that is a `Time` object. We then copy the values of the instance variables of that object to the object we are creating, one at a time. Once we have defined a copy constructor, we can create independent copies like this:

```
Time x = new Time(6, 30, 1.5);
Time y = new Time(x);
```

After this snippet of code has executed, the instance variables for both `x` and `y` will have the same values. But now, we can change `x` and those changes will not effect `y` (and vice versa)!

This might not seem very interesting, and in fact it is not. Writing constructors is a boring, mechanical process. Once you have written several, you will find that you can write them quickly just by looking at the list of instance variables.

11.5 Creating a new object

Although constructors look like methods, you never invoke them directly. Instead, when you invoke `new`, the system allocates space for the new object and then invokes your constructor.

The following program demonstrates two ways to create and initialize `Time` objects:

We've now reached the point where we are writing programs which encompass multiple files. In this case, we have one file, `Time.java` which contains the template for our `Time` objects. This is another example of encapsulation. We have another file which is our startup class and contains our `main`

method.

In order to run, these files must be in the same directory. There are ways to make Java search other directories, but that's more complex then we want to be right now. You only need to compile `Constructors.java`. Java is smart enough to realize that `Time.java` must also be compiled. So Java looks in the same directory, finds a file named `Time.java` and uses that file.

```
public class Time {
    //instance variables
    int hour, minute;
    double second;

    //constructor
    public Time() {
        this.hour = 0;
        this.minute = 0;
        this.second = 0.0;
    }

    //constructor
    public Time(int hour, int minute, double second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    //copy constructor
    public Time(Time t) {
      this.hour = t.hour;
      this.minute = t.minute;
      this.second = t.second;
    }
}
```

Program 11.1: Time.java

```
public class Constructors {
  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);

```

```
10          // another way to do the same thing
11          Time t2 = new Time(11, 8, 3.14159);
12          System.out.println(t2);
13
14          //create an independent copy of t2
15          Time t3 = new Time(t2);
16
17          //notice the values for t2 and t3 are the same
18          System.out.println(t3);
19
20          //change t3
21          t3.hour = 6;
22          System.out.println(t2);
23          System.out.println(t3);
24      }
25  }
```

Program 11.2: Constructors.java

The `Constructors.java` file makes multiple `Time` objects and demonstrates the use of all 3 constructors in the `Time` class.

In `main`, the first time we invoke `new` (line 4 of `Constructors.java`), we provide no arguments, so Java invokes the first constructor on lines 7-11 of `Time.java`. The next few lines assign values to the instance variables.

The second time we invoke `new` (line 11 of `Constructors.java`), we provide arguments that match the parameters of the second constructor (lines 14-18 of `Time.java`). 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.

The last time we invoke new (line 15 of `Constructors.java`), we use the copy constructor (lines 21-25 of `Time.java`) to make a true, independent copy of `t2`. We then change some values of `t3` to demonstrate the fact that `t2` and `t3` are independent objects even though one was created from the other.

11.6 Printing objects

The output of this program is something like:

```
Time@1a7d59b
Time@a6fed5
```

```
Time@1a489ad
Time@a6fed5
Time@1a489ad
```

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.

To print objects in a way that is more meaningful to users (as opposed to programmers), we'll utilize a method that has magic powers in Java.

Every object type has a method called `toString` that returns a string representation of the object. When you print an object using `print` or `println`, Java automatically invokes the object's `toString` method.

The default version of `toString`, which is automatically implemented for you, returns a string that contains the type of the object and a unique identifier like we saw above. When you define a new object type, you can **override** the default behavior by providing a new method with the behavior you want.

In other words, we'll write a method named `toString` which will conform to certain rules. Then Java will automatically invoke this method for us whenever we try to print our objects.

To take advantage of these automatic invocations, our `toString` method must conform to the following rules:

- It must be named `toString`
- It must appear in our object class (`Time.java` in this example)
- It has no parameters
- It returns a `String`
- It should not use the keyword `static`.

So all `toString` methods must have the prototype:

```
1  public String toString()
```

The `String` that is returned should be a human-readable representation of the object's state, as defined by the instance variables. So for the `Time.java` class, a reasonable `toString` method could be:

```
    public String toString() {
        return (this.hour + ":" + this.minute + ":" + this.second);
    }
```

If we add this method to our `Time.java` file, the output becomes:

```
11:8:3.14159
11:8:3.14159
11:8:3.14159
11:8:3.14159
6:8:3.14159
```

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. To make this method's output more standardized, you would need to add some conditional statements which append a '0' to certain parts of the output. You can also do some math which would truncate the seconds portion of the time to 2 digits. This is left as an exercise to the reader (Exercise 11.1).

The `toString` method is an example of an **instance method** which we'll talk more about later in Chapter 12 when we talk more about object-oriented programming.

11.7 Operations on objects

In the next few sections, I demonstrate three kinds of methods that operate on objects:

pure function: Takes objects as arguments but does not modify them. 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.

fill-in method: One of the arguments is an "empty" object that gets filled in by the method. Technically, this is a type of modifier.

Often it is possible to write a given method as a pure function, a modifier, or a fill-in method. I will discuss the pros and cons of each.

11.8 Pure function

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 `isAfter`, which compares two `Time`s and returns a `boolean` that indicates whether the first operand comes after the second:

```
1    public static boolean isAfter(Time time1, Time time2) {
2        if (time1.hour > time2.hour) return true;
3        if (time1.hour < time2.hour) return false;
4
5        if (time1.minute > time2.minute) return true;
6        if (time1.minute < time2.minute) return false;
7
8        if (time1.second > time2.second) return true;
9        return false;
10   }
```

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 `addTime`, which calculates the sum of two times. 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:

```
1    public static Time addTime(Time t1, Time t2) {
2        Time sum = new Time();
3        sum.hour = t1.hour + t2.hour;
4        sum.minute = t1.minute + t2.minute;
5        sum.second = t1.second + t2.second;
6        return sum;
7    }
```

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 = addTime(currentTime, breadTime);
System.out.println(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 corrected version of the method.

```
public static Time addTime(Time t1, Time t2) {
    Time sum = new Time();
    sum.hour = t1.hour + t2.hour;
    sum.minute = t1.minute + t2.minute;
    sum.second = t1.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 it's correct, it's starting to get big. Later I suggest much shorter alternative.

11.9 Modifiers

As an example of a modifier method, consider `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this method looks like:

```
public static void increment(Time time, double secs) {
    time.second += secs;

    if (time.second >= 60.0) {
        time.second -= 60.0;
        time.minute += 1;
    }
    if (time.minute >= 60) {
        time.minute -= 60;
        time.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 replacing the `if` statements with `while` statements:

```
public static void increment(Time time, double secs) {
    time.second += secs;

    while (time.second >= 60.0) {
        time.second -= 60.0;
        time.minute += 1;
    }
    while (time.minute >= 60) {
        time.minute -= 60;
        time.hour += 1;
    }
}
```

This solution is correct, but not very efficient. Can you think of a solution that does not require iteration?

11.10 Fill-in methods

Instead of creating a new object every time `addTime` is invoked, we could require the caller to provide an object where `addTime` stores the result. Compare this to the previous version:

```
    public static void addTimeFill(Time t1, Time t2, Time sum) {
        sum.hour = t1.hour + t2.hour;
        sum.minute = t1.minute + t2.minute;
        sum.second = t1.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;
        }
    }
```

The result is stored in `sum`, so the return type is `void`.

Modifiers and fill-in methods are efficient because they don't have to create new objects. But they make it more difficult to isolate parts of a program; in large projects they can cause errors that are hard to find.

Pure functions help manage the complexity of large projects, in part by making certain kinds of errors impossible. Also, they lend themselves to certain kids of composition and nesting. And because the result of a pure function depends only on the parameters, it is possible to speed them up by storing previously-computed results.

11.11 Incremental development and planning

In this chapter I demonstrated a program development process called **rapid prototyping**[1]. For each method, I wrote a rough draft that performed the basic calculation, then tested it on a few cases, correcting flaws as I found them.

[1]What I am calling rapid prototyping is similar to test-driven development (TDD); the difference is that TDD is usually based on automated testing. See http://en.wikipedia.org/wiki/Test-driven_development.

This approach can be effective, but 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 to look for insight into the problem that can make the programming 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 `addTime` and `increment`, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Another 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 static double convertToSeconds(Time t) {
    int minutes = t.hour * 60 + t.minute;
    double seconds = minutes * 60 + t.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 fourth 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; it involves some calculation along with assignments to the instance variables.

You might have to think to convince yourself that the technique I am using to convert from one base to another is correct. But once you're convinced, we can use these methods to rewrite `addTime`:

```
public static Time addTime(Time t1, Time t2) {
    double seconds = convertToSeconds(t1) + convertToSeconds(t2);
    return new Time(seconds);
}
```

This is 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.

11.12 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 features later. 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 it easier (fewer special cases, fewer opportunities for error).

11.13 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**. This word is not easy to define, so I will try a couple of approaches.

First, consider some things that are not algorithms. 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 learned 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 is 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.

11.14 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.

startup class: The class that contains the `main` method where execution of the program begins.

pure function: A method whose result depends only on its parameters, and that has no side-effects other than returning a value.

modifier: A method that changes one or more of the objects it receives as parameters, and usually returns `void`.

fill-in method: A type of method that takes an "empty" object as a parameter and fills in its instance variables instead of generating a return value.

algorithm: A set of instructions for solving a class of problems by a mechanical process.

11.15 Exercises

Exercise 11.1. In Section ?? we introduced the `toString` method for `Time` objects, but it wasn't very well formatted. Rewrite the `toString` method to return the time in a standard format of hh:mm:ss.

Exercise 11.2. In the board game Scrabble[2], each tile contains a letter, which is used to spell words, and a score, which is used to determine the value of words.

1. Write a definition for a class named `Tile` that represents Scrabble tiles. The instance variables should be a character named `letter` and an integer named `value`.

2. Write a constructor that takes parameters named `letter` and `value` and initializes the instance variables.

3. Write a method named `printTile` that takes a `Tile` object as a parameter and prints the instance variables in a reader-friendly format.

4. Write a method named `testTile` that creates a Tile object with the letter `Z` and the value 10, and then uses `printTile` to print the state of the object.

The point of this exercise is to practice the mechanical part of creating a new class definition and code that tests it.

Exercise 11.3. Write a class definition for `Date`, an object type that contains three integers, `year`, `month` and `day`. This class should provide two constructors. The first should take no parameters. The second should take parameters named `year`, `month` and `day`, and use them to initialize the instance variables.

Write a `main` method that creates a new `Date` object named `birthday`. The new object should contain your birthdate. You can use either constructor.

[2]Scrabble is a registered trademark owned in the U.S.A and Canada by Hasbro Inc., and in the rest of the world by J.W. Spear & Sons Limited of Maidenhead, Berkshire, England, a subsidiary of Mattel Inc.

Exercise 11.4. A rational number is a number that can be represented as the ratio of two integers. For example, 2/3 is a rational number, and you can think of 7 as a rational number with an implicit 1 in the denominator. For this assignment, you are going to write a class definition for rational numbers.

1. Create a new program called `Rational.java` that defines a class named `Rational`. A `Rational` object should have two integer instance variables to store the numerator and denominator.

2. Write a constructor that takes no arguments and that sets the numerator to 0 and denominator to 1.

3. Write a method called `printRational` that takes a Rational object as an argument and prints it in some reasonable format.

4. Write a `main` method that creates a new object with type Rational, sets its instance variables to some values, and prints the object.

5. At this stage, you have a minimal testable program. Test it and, if necessary, debug it.

6. Write a second constructor for your class that takes two arguments and that uses them to initalize the instance variables.

7. Write a method called `negate` that reverses the sign of a rational number. This method should be a modifier, so it should return `void`. Add lines to `main` to test the new method.

8. Write a method called `invert` that inverts the number by swapping the numerator and denominator. Add lines to `main` to test the new method.

9. Write a method called `toDouble` that converts the rational number to a double (floating-point number) and returns the result. This method is a pure function; it does not modify the object. As always, test the new method.

10. Write a modifier named `reduce` that reduces a rational number to its lowest terms by finding the greatest common divisor (GCD) of the numerator and denominator and dividing through. This method should

be a pure function; it should not modify the instance variables of the object on which it is invoked. To find the GCD, see Exercise 5.8).

11. Write a method called `add` that takes two Rational numbers as arguments and returns a new Rational object. The return object should contain the sum of the arguments.

    There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).

The purpose of this exercise is to write a class definition that includes a variety of methods, including constructors, modifiers and pure functions.

Chapter 12

Object-oriented programming

12.1 Programming languages and styles

There are many programming languages and almost as many programming styles (sometimes called paradigms). The programs we have written so far are **procedural**, because the emphasis has been on specifying computational procedures.

Most Java programs are **object-oriented**, which means that the focus is on objects and their interactions. Here are some of the characteristics of object-oriented programming:

- Objects often represent entities in the real world. In the next few chapters, we will model cards and collections of cards (decks or hands).

- The majority of methods are instance methods (like the methods you invoke on `Strings`) rather than class methods (like the `Math` methods). Most of the methods we have written so far have been class methods. In this chapter we write some instance methods.

- Objects are isolated from each other by limiting the ways they interact, especially by preventing them from accessing instance variables without invoking methods.

12.2 Instance methods and class methods

There are two types of methods in Java, called **class methods** and **instance methods** (sometimes also called "object methods"). Class methods are identified by the keyword `static` in the first line. Any method that does *not* have the keyword `static` is an instance method.

Although we have not written many instance methods, we have invoked some. Whenever you invoke a method "on" an object, it's an instance method. For example, `charAt` and the other methods we invoked on `String` objects are all instance methods.

Actually, the `toString` method we learned about in Section 11.6 is an instance method, but we didn't call it that at that point. Now, we'll dive a bit more into instance vs. class methods. But first, let's set the stage and look at a real world object we can model using objects and classes: Cards and decks of cards.

12.3 `Card` objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. Or read http://en.wikipedia.org/wiki/Playing_card.

There are 52 cards in a deck; each belongs to one of four suits and one of 13 ranks. The suits are Spades, Hearts, Diamonds and Clubs (in descending order in Bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King. Depending on what game you are playing, the Ace may be considered higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is pretty obvious what the instance variables should be: `rank` and `suit`. It is not as obvious what type the instance variables should be. One possibility is `String`s, containing things like `"Spade"` for suits and `"Queen"` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By "encode" I do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by "encode" is something like "define a mapping between a sequence of numbers and the things I want to represent." For example,

Spades $\mapsto$ 3
Hearts $\mapsto$ 2
Diamonds $\mapsto$ 1
Clubs $\mapsto$ 0

The obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack $\mapsto$ 11
Queen $\mapsto$ 12
King $\mapsto$ 13

The reason I am using mathematical notation for these mappings is that they are not part of the program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```
class Card
{
    int suit, rank;

    public Card() {
        this.suit = 0;  this.rank = 0;
    }

    public Card(int suit, int rank) {
        this.suit = suit;  this.rank = rank;
    }
}
```

As usual, I provide two constructors: one takes a parameter for each instance variable; the other takes no parameters.

To create an object that represents the 3 of Clubs, we invoke `new`:

```
    Card threeOfClubs = new Card(0, 3);
```

The first argument, `0` represents the suit Clubs.

12.4 The `toString` method

When you create a new class, the first step is to declare the instance variables and write constructors. The second step is to write the standard methods that every object should have, including the `toString` instance method which

lets us easily print the object.

To print `Card` objects in a way that humans can read easily, we want to map the integer codes onto words. A natural way to do that is with an array of `String`s. You can create an array of `String`s the same way you create an array of primitive types:

```
String[] suits = new String[4];
```

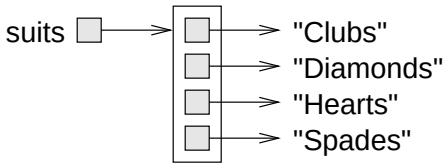
Then we can set the values of the elements of the array.

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

Creating an array and initializing the elements is such a common operation that Java provides a special syntax for it:

```
String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

This statement is equivalent to the separate declaration, allocation, and assignment. The state diagram of this array looks like:



The elements of the array are *references* to the `String`s, rather than `String`s themselves.

Now we need another array of `String`s to decode the ranks:

```
String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
           "7", "8", "9", "10", "Jack", "Queen", "King" };
```

The reason for the `"narf"` is to act as a place-keeper for the zeroeth element of the array, which is never used (or shouldn't be). The only valid ranks are 1–13. To avoid this wasted element, we could have started at 0, but the mapping is more natural if we encode 2 as 2, and 3 as 3, etc.

Using these arrays, we can select the appropriate `String`s by using the `suit` and `rank` as indices. We can add the following `toString` method to our `Card` class:

```
public String toString() {
    String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
```

```
                "7", "8", "9", "10", "Jack", "Queen", "King" };
    return ranks[this.rank] + " of " + suits[this.suit];
}
```

the expression `suits[this.suit]` means "use the instance variable `suit` from the current object `this` as an index into the array named `suits`, and select the appropriate string." The output of this code

```
    Card card = new Card(1, 11);
    System.out.println( card.toString() );
```

is `Jack of Diamonds`.

To invoke an instance method, you usually must invoke it *on* an object like we did above. However, Java actually creates some shortcuts for us using the `toString method`. If this method is implemented, Java will use the `String` that is returned if your object is treated like a String. For example, the code:

```
    Card card = new Card(1, 11);
    System.out.println("my card is " +  card);
```

doesn't really make sense. How do you concatenate a `Card` object with a `String`? You can't; Java requires that types match for operations and a `String` and `Card` aren't the same. However, since we've implemented the `toString` method, Java can figure out how to treat our object like a String! In fact, Java will do this automatically for us even if we don't invoke the `toString` method explicitly. So we can write code like

```
    System.out.println("my card is " +  card);
```

or just

```
    System.out.println(card);
```

without explicitly invoking the `toString` method on our object. However, that's what Java is doing for us behind the scenes.

In fact, every object type has a method called `toString` that returns a string representation of the object. The default version of `toString` returns a string that contains the type of the object and a unique identifier (see Section 11.6). When you define a new object type, you can **override** the default behavior like we did above by providing a new method with the behavior you want.

12.5 The sameCard method

Anything that can be written as a class method can also be written as an instance method, and vice versa. But sometimes it is more natural to use one or the other. To see this concept in action, let's think about determine whether two cards are the same or not.

The word "same" is one of those things that occur in natural language that seem perfectly clear until you give it some thought, and then you realize there is more to it than you expected.

For example, if I say "Chris and I have the same car," I mean that his car and mine are the same make and model, but they are two different cars. If I say "Chris and I have the same mother," I mean that his mother and mine are one person. So the idea of "sameness" is different depending on the context.

When you talk about objects, there is a similar ambiguity. For example, if two Cards are the same, does that mean they contain the same data (rank and suit), or they are actually the same Card object?

To see if two references refer to the same object, we use the == operator. For example:

```
Card card1 = new Card(1, 11);
Card card2 = card1;

if (card1 == card2) {
    System.out.println("card1 and card2 are identical.");
}
```

References to the same object are **identical**. References to objects with same data are **equivalent**.

To check equivalence, it is common to write a class method with a name like sameCard.

```
public static boolean sameCard(Card c1, Card c2) {
    return(c1.suit == c2.suit && c1.rank == c2.rank);
}
```

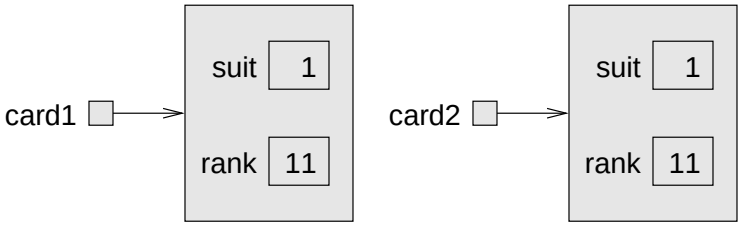
Here is an example that creates two objects with the same data, and uses sameCard to see if they are equivalent:

```
Card card1 = new Card(1, 11);
Card card2 = new Card(1, 11);

if (sameCard(card1, card2)) {
```

```
        System.out.println("card1 and card2 are equivalent.");
    }
```

If references are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

In this case, `card1` and `card2` are equivalent but not identical, so the state diagram looks like this:



What does it look like when `card1` and `card2` are identical?

In Section 8.12 I said that you should not use the `==` operator on `String`s because it does not do what you expect. Instead of comparing the contents of the `String` (equivalence), it checks whether the two `String`s are the same object (identity).

Now let's implement this method as an instance method instead of a class method. In the instance method, we'll compare the current `Card` object (the one the instance method is invoked on) to a `Card` passed to the method as an argument. Here is the method re-written as an instance method:

```
    public boolean sameAs(Card c) {
        return(this.suit == c.suit && this.rank == c.rank);
    }
```

Here are the changes:

1. I removed `static` from the method header.

2. I changed the name of the method to be more idiomatic.

3. I removed one parameter.

4. Inside an object method you can refer to instance variables as if they were local variables, so I changed `c1.rank` to `this.rank`, and likewise for `suit`.

Here's how this method is invoked:

```
Card c1 = new Card(1, 11);
Card c2 = new Card(1, 11);
if (c1.sameAs(c2)) {
  System.out.println("card 1 and card2 are equivalent");
}
```

When you invoke a method on an object, that object becomes the **current object**, also known as `this`. Inside `sameAs`, the keyword `this` refers to the card the method was invoked on.

12.6 The `equals` method

In Section 12.5 we talked about two notions of equality: identity, which means that two variables refer to the same object, and equivalence, which means that they have the same value.

The `==` operator tests identity, but there is no operator that tests equivalence, because what "equivalence" means depends on the type of the objects. When working with the `String` class, we learned that we had to use the `equals` method to compare two tt Strings. In general, we should implement a method named `equals` if we want to compare two objects.

Java classes provide `equals` methods that do the right thing. But for user defined types the default behavior is the same as identity, which is usually not what you want.

For `Card`s we already have an instance method that checks equivalence:

```
public boolean sameAs(Card c) {
    return (this.suit == c.suit && this.rank == c.rank);
}
```

So all we have to do is change the name to follow conventions:

```
public boolean equals(Card c) {
    return (suit == c.suit && rank == c.rank);
}
```

Here's how it's invoked:

```
Card card = new Card(1, 1);
Card card2 = new Card(1, 1);
System.out.println(card.equals(card2));
```

Inside `equals`, `this` is the current object and `c` is the parameter. For methods that operate on two objects of the same type, I sometimes use `this` explicitly

and call the parameter `that`:

```
1    public boolean equals(Card that) {
2        return (this.suit == that.suit && this.rank == that.rank);
3    }
```

I think it improves readability.

12.7 Oddities and errors

If you have instance methods and class methods in the same class, it is easy to get confused. A common way to organize a class definition is to put all the constructors at the beginning, followed by all the object methods and then all the class methods.

You can have an object method and a class method with the same name, as long as they do not have the same number and types of parameters. As with other kinds of overloading, Java decides which version to invoke by looking at the arguments you provide.

Now that we know what the keyword `static` means, you have probably figured out that `main` is a class method, which means that there is no "current object" when it is invoked. Since there is no current object in a class method, it is an error to use the keyword `this`. If you try, you get an error message like: "Undefined variable: this."

Also, you cannot refer to instance variables without using dot notation and providing an object name. If you try, you get a message like "non-static variable... cannot be referenced from a static context." By "non-static variable" it means "instance variable."

12.8 The `compareCard` method

For primitive types, the conditional operators compare values and determine when one is greater or less than another. These operators (`<` and `>` and the others) don't work for object types. For `String`s Java provides a `compareTo` method. For `Card`s we have to write our own, which we will call `compareTo` as well.

Some sets are completely ordered, which means that you can compare any two elements and tell which is bigger. Integers and floating-point numbers are totally ordered. Some sets are unordered, which means that there is no

meaningful way to say that one element is bigger than another. Fruits are unordered, which is why we cannot compare apples and oranges. In Java, the `boolean` type is unordered; we cannot say that `true` is greater than `false`.

The set of playing cards is partially ordered, which means that sometimes we can compare cards and sometimes not. For example, I know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

To make cards comparable, we have to decide which is more important, rank or suit. The choice is arbitrary, but when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on. So let's say that suit is more important.

With that decided, we can write the `compareTo` instance method. It should compare the current `Card` to a parameter `Card` and return 1 if the current card wins (is "higher" or "better"), -1 if the parameter card wins, and 0 if they are equivalent.

First we compare suits:

```
1    if (this.suit > c.suit) return 1;
2    if (this.suit < c.suit) return -1;
```

If neither statement is true, the suits must be equal, and we have to compare ranks:

```
1    if (this.rank > c.rank) return 1;
2    if (this.rank < c.rank) return -1;
```

If neither of these is true, the ranks must be equal, so we return `0`.

Putting it all together, we have:

```
1  public int compareTo(Card c) {
2    if (this.suit > c.suit) return 1;
3    if (this.suit < c.suit) return -1;
4    if (this.rank > c.rank) return 1;
5    if (this.rank < c.rank) return -1;
6    return 0;
```

12.9 Wrapping up

You can download the entire `Card` class from:
http://www.mathcs.emory.edu/~valerie/textbook/programs/Card.java

You can write a small program which makes `Card` objects experiment with invoking various instance methods them.

12.10 Glossary

class method: A method with the keyword `static`. Class methods are not invoked on objects and they do not have a current object.

current object: The object on which an instance method is invoked. Inside the method, the current object is referred to by `this`.

encode: To represent one set of values using another set of values, by constructing a mapping between them.

equivalence: Equality of values. Two references that point to objects that contain the same data.

explicit: Anything that is spelled out completely. Within a class method, all references to the instance variables have to be explicit.

identity: Equality of references. Two references that point to the same object in memory.

implicit: Anything that is left unsaid or implied. Within an instance method, you can refer to the instance variables implicitly (i.e., without naming the object).

instance method: A method that is invoked on an object, and that operates on that object. Instance methods do not have the keyword `static`.

12.11 Exercises

Exercise 12.1. Modify the code in the instance method `compareTo` in the `Card` class so that aces are ranked higher than Kings. For example an Ace of spaces would be higher than a King of Spades.

Exercise 12.2. Transform the following class method into an instance method for the `Complex` class. Note that you really don't have to understand what the `Complex` class is or what the math below does.

```
public static double abs(Complex c) {
    return Math.sqrt(c.real * c.real + c.imag * c.imag);
}
```

Exercise 12.3. Transform the following instance method into a class method.

```
public boolean equals(Complex b) {
    return(real == b.real && imag == b.imag);
}
```

Exercise 12.4. This exercise is a continuation of Exercise 11.4. The purpose is to practice the syntax of object methods and get familiar with the relevant error messages.

1. Transform the methods in the `Rational` class from class methods to object methods, and make the necessary changes in `main`.

2. Make a few mistakes. Try invoking class methods as if they were object methods and vice-versa. Try to get a sense for what is legal and what is not, and for the error messages that you get when you mess up.

3. Think about the pros and cons of class and object methods. Which is more concise (usually)? Which is a more natural way to express computation (or, maybe more fairly, what kind of computations can be expressed most naturally using each style)?

Chapter 13

Arrays of Objects

Now that we have `Card` objects, we can do interesting things with them. Before we dive in, here is an outline of the steps:

1. We'll now group objects together into an array to form a basic `Deck` of cards.

2. Then, we'll extend this idea and will create a `Deck` class and write methods that operate on `Deck`s.

13.1 Arrays of cards

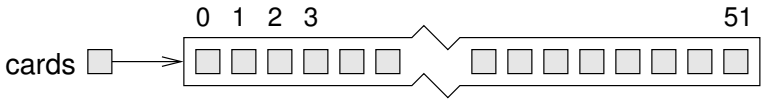
By now we have seen several examples of composition (the ability to combine language features in a variety of arrangements). One of the first examples we saw was using a method invocation as part of an expression. Another example is the nested structure of statements: you can put an `if` statement within a `while` loop, or within another `if` statement, etc.

Having seen this pattern, and having learned about arrays and objects, you should not be surprised to learn that you can make arrays of objects. And you can define objects with arrays as instance variables; you can make arrays that contain arrays; you can define objects that contain objects, and so on. Now we'll see examples of these combinations using `Card` objects.

This example creates an array of 52 cards:

```
1    Card[] cards = new Card[52];
```

Here is the state diagram for this object:



The array contains *references* to objects; it does not contain the `Card` objects themselves. The elements are initialized to `null`. You can access the elements of the array in the usual way:

```
if (cards[0] == null) {
    System.out.println("No cards yet!");
}
```

But if you try to access the instance variables of the non-existent `Card`s, you get a `NullPointerException`.

```
cards[0].rank;             // NullPointerException
```

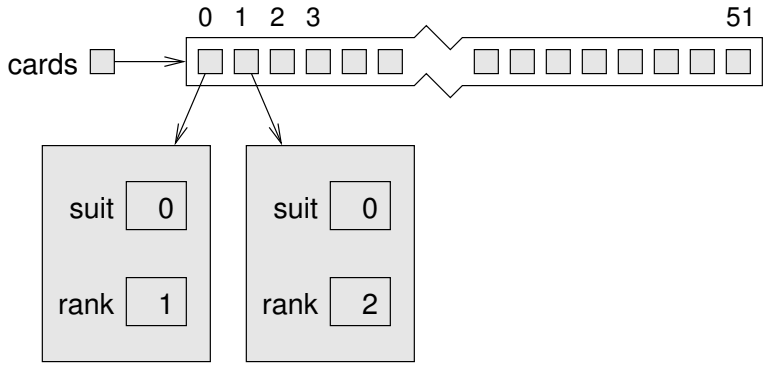
But that is the correct syntax for accessing the `rank` of the "zeroeth" card in the deck. This is another example of composition, combining the syntax for accessing an element of an array and an instance variable of an object.

The easiest way to populate the deck with `Card` objects is to write nested for loops (i.e., one loop inside the body of another):

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
        cards[index] = new Card(suit, rank);
        index++;
    }
}
```

The outer loop enumerates the suits from 0 to 3. For each suit, the inner loop enumerates the ranks from 1 to 13. Since the outer loop runs 4 times, and the inner loop runs 13 times, the body is executed is 52 times.

I used `index` to keep track of where in the deck the next card should go. The following state diagram shows what the deck looks like after the first two cards have been allocated:



Now that we have an idea of how we can group Cards together using an array, we can wrap our deck of Cards up into a new object called Deck and add some standard behaviors like shuffling a deck.

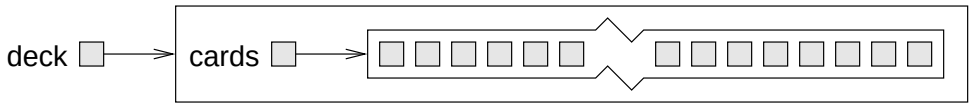
13.2 The Deck class

We can now create Deck objects. Each Deck should have 52 Card objects in it (contained in an array which will be an instance variable).

The class definition looks like this:

```
class Deck {
    Card[] cards;

    public Deck(int n) {
        this.cards = new Card[n];
    }
}
```

The constructor initializes the instance variable with an array of cards, but it doesn't create any cards. Our array will be full of null objects, which isn't ideal. Here is a state diagram showing what a Deck looks like with no cards:



Here is a no-argument constructor that makes a 52-card deck and populates it with Cards:

```
    public Deck() {
        this.cards = new Card[52];
        int index = 0;
        for (int suit = 0; suit <= 3; suit++) {
```

```
            for (int rank = 1; rank <= 13; rank++) {
                cards[index] = new Card(suit, rank);
                index++;
            }
        }
    }
```

To invoke it, we use `new`:

```
    Deck deck = new Deck();
```

13.3 The `toString` method

Now it makes sense to put the methods that pertain to `Deck`s in the `Deck` class definition. Looking at the methods we have written so far, one obvious candidate is `toString` (Section 12.4). Here's how it looks, rewritten to work with a `Deck`:

```
    public String toString() {
        String d = "";
        for (int i = 0; i < cards.length; i++) {
            d = d + cards[i].toString() + "\n";
        }
        return d;
    }
```

We use the `Deck` class's instance variable `cards` to iterate over all the `Card` objects in our deck. We then use `cards[i]` to access an element of the array. We can use the `toString` representation that we built into our `Card` class (Section 12.4)! We build up a `String` representation of our `Deck` object by using a `String` representation of each `Card` in the deck!

13.4 Shuffling

For most card games you need to be able to shuffle the deck; that is, put the cards in a random order. In Section 9.7 we saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle, which is usually by dividing the deck in two and then choosing alternately from each deck. Since humans usually don't shuffle perfectly, after about 7 iterations the

order of the deck is pretty well randomized. But a computer program would have the annoying property of doing a perfect shuffle every time, which is not really very random. In fact, after 8 perfect shuffles, you would find the deck back in the order you started in. For more information, see http://en.wikipedia.org/wiki/Faro_shuffle.

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration choose two cards at random and swap them.

Here is an outline of how this algorithm works. To sketch the program, I am using a combination of Java statements and English words that is sometimes called **pseudocode**:

```
    for (int i = 0; i < cards.length; i++) {
        // choose a number between i and cards.length-1
        // swap the ith card and the randomly-chosen card
    }
```

The nice thing about pseudocode is that it often makes it clear what methods you are going to need. In this case, we will need to generate random integer between `0` and `cards.length`, and code to swap two `Cards` in the `cards` array.

This process—writing pseudocode first and then writing methods to make it work—is called **top-down development** (see http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design).

Putting the pieces together, our instance method will look like:

```
  public void shuffle() {
    for (int i = 0; i < cards.length; i++) {
      int random = (int)(Math.random() * cards.length)
      Card temp = cards[i];
      cards[i] = cards[random];
      cards[random] = temp;
    }
  }
```

To invoke this method, we will need some code like:

```
  Deck deck = new Deck();
  System.out.println( deck );
  deck.shuffle();
  System.out.println("After shuffle: ");
  System.out.println( deck );
```

When we execute this code, we will see something like:

```
  Ace of Clubs
```

```
  2 of Clubs
  3 of Clubs
  ... rest of sorted deck omitted ...
  After shuffle:
  4 of Clubs
  4 of Diamonds
  3 of Hearts
  Ace of Hearts
  ... rest of shuffled deck omitted ...
```

13.5 Searching

The next instance method I'll write is `findCard`, which searches an array of `Card`s in a `Deck` to see whether it contains a certain card. This method gives us a chance to utilize two searching algorithms we previously learned about: **linear search** and **binary search**.

Linear search is pretty obvious; we traverse the deck and compare each card to the one we are looking for. If we find it we return the index where the card appears. If it is not in the deck, we return -1.

```
public int findCard(Card card) {
    for (int i = 0; i < cards.length; i++) {
        if (card.equals(cards[i])) {
            return i;
        }
    }
    return -1;
}
```

The argument of `findCard` are a single `Card` object, `card`. It might seem odd to have a variable with the same name as a type (the `card` variable has type `Card`). We can tell the difference because the variable begins with a lower-case letter.

The method returns as soon as it discovers the card, which means that we do not have to traverse the entire deck if we find the card we are looking for. If we get to the end of the loop, we know the card is not in the deck.

If the cards in the deck are not in order, there is no way to search faster than this. We have to look at every card because otherwise we can't be certain the card we want is not there.

But when you look for a word in a dictionary, you don't search linearly

through every word, because the words are in alphabetical order. As a result, you probably use an algorithm similar to a binary search:

1. Start in the middle somewhere.

2. Choose a word on the page and compare it to the word you are looking for.

3. If you find the word you are looking for, stop.

4. If the word you are looking for comes after the word on the page, flip to somewhere later in the dictionary and go to step 2.

5. If the word you are looking for comes before the word on the page, flip to somewhere earlier in the dictionary and go to step 2.

If you ever get to the point where there are two adjacent words on the page and your word comes between them, you can conclude that your word is not in the dictionary.

Getting back to the deck of cards, if we know the cards are in order, we can write a faster version of `findCard`. The best way to write a binary search is with a recursive method, because it is naturally recursive.

The trick is to write a method called `findBinary` that takes two indices as parameters, `low` and `high`, indicating the segment of the array that should be searched (including both `low` and `high`).

1. To search the array, choose an index between `low` and `high` (call it `mid`) and compare it to the card you are looking for.

2. If you found it, stop.

3. If the card at `mid` is higher than your card, search the range from `low` to `mid-1`.

4. If the card at `mid` is lower than your card, search the range from `mid+1` to `high`.

Steps 3 and 4 look suspiciously like recursive invocations. Here's what this looks like translated into Java code:

```
public int findBinary(Card card, int low, int high) {
    // TODO: need a base case
    int mid = (high + low) / 2;
    int comp = cards[mid].compareTo(card);

    if (comp == 0) {
        return mid;
    } else if (comp > 0) {
        return findBinary(card, low, mid-1);
    } else {
        return findBinary(card, mid+1, high);
    }
}
```

This code contains the kernel of a binary search, but it is still missing an important piece, which is why I added a TODO comment. As written, the method recurses forever if the card is not in the deck. We need a base case to handle this condition.

If `high` is less than `low`, there are no cards between them, see we conclude that the card is not in the deck. If we handle that case, the method works correctly:

```
public int findBinary(Card card, int low, int high) {
    System.out.println(low + ", " + high);

    if (high < low) return -1;

    int mid = (high + low) / 2;
    int comp = cards[mid].compareTo(card);

    if (comp == 0) {
        return mid;
    } else if (comp > 0) {
        return findBinary(card, low, mid-1);
    } else {
        return findBinary(card, mid+1, high);
    }
}
```

I added a print statement so I can follow the sequence of recursive invocations. I tried out the following code:

```
    Deck deck = new Deck();
    Card card1 = new Card(1, 11);
    System.out.println(deck.findBinary(card1, 0, 51));
```

And got the following output:

```
0, 51
0, 24
13, 24
19, 24
22, 24
23
```

Then I made up a card that is not in the deck (the 15 of Diamonds), and tried to find it. I got the following:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1
```

These tests don't prove that this program is correct. In fact, no amount of testing can prove that a program is correct. On the other hand, by looking at a few cases and examining the code, you might be able to convince yourself.

The number of recursive invocations is typically 6 or 7, so we only invoke `compareCard` 6 or 7 times, compared to up to 52 times if we did a linear search. In general, binary search is much faster than a linear search, and even more so for large arrays.

Two common errors in recursive programs are forgetting to include a base case and writing the recursive call so that the base case is never reached. Either error causes infinite recursion, which throws a `StackOverflowException`. (Think of a stack diagram for a recursive method that never ends.)

13.6 Decks and subdecks

Here is the header or signature (see Section 8.5) of `findBinary`:

```
public int findBinary(Card card, int low, int high)
```

We can think of the instance variable `cards`, and the parameter variables `low`, and `high` as a single parameter that specifies a **subdeck**. This way of thinking is common, and is sometimes referred to as an **abstract parameter**. What I mean by "abstract" is something that is not literally part of the program text, but which describes the function of the program at a higher level.

For example, when you invoke the method and pass the bounds `low` and `high`, there is nothing that prevents the invoked method from accessing parts of the array that are out of bounds. So you are not literally using a subset of the deck; you are really using the whole deck. But as long as the recipient plays by the rules, it makes sense to think of it abstractly as a subdeck.

This kind of thinking, in which a program takes on meaning beyond what is literally encoded, is an important part of thinking like a computer scientist. The word "abstract" gets used so often and in so many contexts that it comes to lose its meaning. Nevertheless, **abstraction** is a central idea in computer science (and many other fields).

A more general definition of "abstraction" is "The process of modeling a complex system with a simplified description to suppress unnecessary details while capturing relevant behavior."

At other times, it may be useful to explicitly create a subdeck. For example, how should we represent a hand or some other subset of a full deck? One possibility is to create a new class called `Hand`. Another possibility, the one I will demonstrate, is to represent a hand with a `Deck` object with fewer than 52 cards.

We might want a method, `subdeck`, that returns a new Deck that contains a specified subset of the cards:

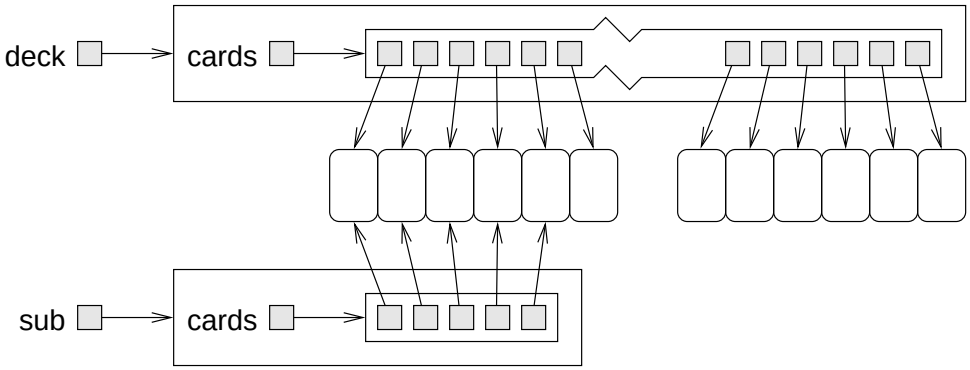
```
public Deck subdeck(int low, int high) {
    Deck sub = new Deck(high-low+1);

    for (int i = 0; i < sub.cards.length; i++) {
        sub.cards[i] = cards[low+i];
    }
    return sub;
}
```

The length of the subdeck is `high-low+1` because both the low card and high card are included. This sort of computation can be confusing, and lead

to "off-by-one" errors. Drawing a picture is usually the best way to avoid them.

Because we provide an argument with `new`, the constructor that gets invoked will be the first one in our `Deck` class, which only allocates the array and doesn't allocate any cards. Inside the `for` loop, the subdeck gets populated with copies of the references from the deck.

The following is a state diagram of a subdeck being created with the parameters `low=3` and `high=7`. The result is a hand with 5 cards that are shared with the original deck; i.e. they are aliased.



Aliasing is usually not generally a good idea, because changes in one subdeck are reflected in others, which is not the behavior you would expect from real cards and decks. But if the cards are immutable, aliasing is less dangerous. In this case, there is probably no reason ever to change the rank or suit of a card. Instead we can create each card once and then treat it as an immutable object. So for `Card`s aliasing is a reasonable choice.

13.7 Shuffling and dealing

In Section 13.4 I wrote pseudocode for a shuffling algorithm. Assuming that we have a method called `shuffle` that takes a deck as an argument and shuffles it, we can use it to deal hands:

```
Deck deck = new Deck();
deck.shuffle();

Deck hand1 = deck.subdeck(0, 4);
Deck hand2 = deck.subdeck(5, 9);
Deck pack = deck.subdeck(10, 51);
```

This code puts the first 5 cards in one hand, the next 5 cards in the other, and the rest into the pack.

When you thought about dealing, did you think we should give one card to each player in the round-robin style that is common in real card games? I thought about it, but then realized that it is unnecessary for a computer program. The round-robin convention is intended to mitigate imperfect shuffling and make it more difficult for the dealer to cheat. Neither of these is an issue for a computer.

This example is a useful reminder of one of the dangers of engineering metaphors: sometimes we impose restrictions on computers that are unnecessary, or expect capabilities that are lacking, because we unthinkingly extend a metaphor past its breaking point.

13.8 A last note on class variables

In our `Card` and `Deck` classes, we have used local variables, which are declared inside a method, parameter variables, which are declared as part of method headers, and instance variables, which are declared in a class definition, usually before the method definitions.

Local variables are created when the are defined and destroyed when they go out of scope. Parameter variables are created when a method is invoked and destroyed when the method ends. Instance variables are created when you create an object and destroyed when the object is garbage collected.

Now lets look at the role class **class variables** can play in object oriented programming. Like instance variables, class variables are defined in a class definition before the method definitions, but they are identified by the keyword `static`. They are created when the program starts and survive until the program ends.

You can refer to a class variable from anywhere inside the class definition. Class variables are often used to store constant values that are needed in several places.

In the case of the `Card` class, the variables `suits` and `ranks`, used in the `toString` method are good candidates for class variables instead of local variables. Presumably, all `Card` objects will use the same mappings for numbers to suits/ranks. Therefore, we only need one set of these variables, and all `Card` objects can use the same mappings.

As an example, here is a version of `Card` where `suits` and `ranks` are class variables:

```
class Card {
    int suit, rank;

    static String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    static String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                      "7", "8", "9", "10", "Jack", "Queen", "King" };

    public String toString() {
        return ranks[rank] + " of " + suits[suit]);
    }
}
```

Inside `toString` we can refer to `suits` and `ranks` as if they were local variables because there are no local or parameter variables to shadow the class variables.

13.9 Wrapping up

You can download the entire `Deck` class from:
http://www.mathcs.emory.edu/~valerie/textbook/programs/Deck.java
Note that this file and the `Card` class (found at the end of Chapter 12) need to be in the same directory if you want to use them.

You can write a small program which makes a `Deck` of cards and experiment with invoking various instance methods on the deck.

13.10 Glossary

abstract parameter: A set of parameters that act together as a single parameter.

abstraction: The process of interpreting a program (or anything else) at a higher level than what is literally represented by the code.

class variable: A variable declared within a class as `static`; there is always exactly one copy of this variable in existence.

pseudocode: A way of designing programs by writing rough drafts in a combination of English and Java.

13.11 Exercises

Exercise 13.1. In Blackjack the object of the game is to get a collection of cards with a score of 21. The score for a hand is the sum of scores for all cards. The score for an aces is 1, for all face cards is ten, and for all other cards the score is the same as the rank. Example: the hand (Ace, 10, Jack, 3) has a total score of 1 + 10 + 10 + 3 = 24.

Write a instance method called `score` which can be invoked on a `Deck` of cards (representing a single hand) which calculates and returns the total score for the cards in the `Deck`.

Exercise 13.2. In Poker a "flush" is a hand that contains five or more cards of the same suit. A hand can contain any number of cards.

1. Write an instance method for the `Deck` class called `suitHist` that takes an returns a an array which is a histogram of the suits in the hand/deck. Your solution should only traverse the array once.

2. Write an instance method for the `Deck` class called `isFlush` that returns `true` if the deck/hand contains a flush, and `false` otherwise.

Exercise 13.3. The goal of this exercise is to write a program that generates random poker hands and classifies them, so that we can estimate the probability of the various poker hands. If you don't play poker, you can read about it here http://en.wikipedia.org/wiki/List_of_poker_hands.

1. Start with
   http://www.mathcs.emory.edu/~valerie/textbook/programs/Card.java
   and http://www.mathcs.emory.edu/~valerie/textbook/programs/Deck.java
   Write a small program named `PlayPoker.java` which creates a deck of cards.

2. Write a definition for a class named `PokerHand` which is composed of an arbitrary number of `Cards`.

3. Write a `Deck` method named `deal` that creates a PokerHand, transfers cards from the deck to the hand, and returns the hand.

4. In `main` use `shuffle` and `deal` to generate and print four `PokerHands` with five cards each. Did you get anything good?

5. Write a `PokerHand` method called `hasFlush` returns a boolean indicating whether the hand contains a flush.

6. Write a method called `hasThreeKind` that indicates whether the hand contains three of a kind (3 cards of the same rank).

7. Write a loop that generates a few thousand hands and checks whether they contain a flush or three of a kind. Estimate the probability of getting one of those hands. Compare your results to the probabilities at http://en.wikipedia.org/wiki/List_of_poker_hands.

8. Write methods that test for the other poker hands. Some are easier than others. You might find it useful to write some general-purpose helper methods that can be used for more than one test.

9. In some poker games, players get seven cards each, and they form a hand with the best five of the seven. Modify your program to generate seven-card hands and recompute the probabilities.

Appendix A

Setting up Your Computer

A.1 Overview

All students enrolled in this course will have access to the lab computers in room MSC E308A seven days a week, during operating hours. You can always check the hours at
http://www.mathcs.emory.edu/computinglab.php.
These computers are already pre-configured with all the software necessary for this course and all files that are saved onto them are regularly backed-up to Emory's servers, leaving no risk of data loss. Students will always have access to one of these lab computers for lab session.

It is not necessary for students to have their own desktop or laptop to be successful in this course, but those who do and want to work on class assignments from their personal machine have two options in addition to using the lab computers:

- Remote Log-in
- Self-install

This semester both options will be supported by the UTAs if you have difficulties working on assignments or setting up your computer, but in using either of these solutions there are some things to be aware of:

- Both Remote Log-in and Self-install will require software to be installed on your computer, and it will be your responsibility to maintain the software by installing any security updates required.

- Remote Log-in will require an active and constant (relatively high speed) internet connection, and it is preferable that students live or work primarily on-campus.

- Self-install will leave you with copies of all assignments on your local machine only until submission. Unlike Remote Log-in, where all files are backed up for you, Self-install leaves students open to the loss of their assignments in the event of a crash or other computer failure. Data loss due to a crash or other computer issue will never be accepted as an excuse for a late assignment in this course. This can be avoided by regularly backing up your files if you opt to use the Self-install option and we provide instructions below.

A.2 Choosing an option

There are pluses and minuses to both option. This section is intended to help you decide which option is better for you.

Remote Log-in allows you to log into a lab computer remotely through SSH, (Secure Shell) and X11. You are essentially working on a lab computer to work and using your computers screen as a display. While logged in, you will be able to access all files and applications that are available to you when physically using a lab computer. This option tends to run slightly slower than Self-install because all commands or requests (such as typing or compiling a file) go through the internet and are not performed on your personal computer. This also means that when not logged in or unable to connect to internet, you will not be able to access, view or edit any files that are saved on Emory's lab computers.

Self-install has you install all the software needed to complete work for this course on your personal computer. As discussed above, this means the only copy of all files are stored on your personal computer and not on Emory's computers. However Self-install allows you to work online or offline and is generally more responsive/faster than Remote Log-in. This is a better option for most students as long as you back up your content regularly.

The next sections will walk you through setting up either of these options on a Mac or PC. If you have any issues while attempting to setup your computer, post on Piazza first and if you are still having issues, stop by UTA office hours.

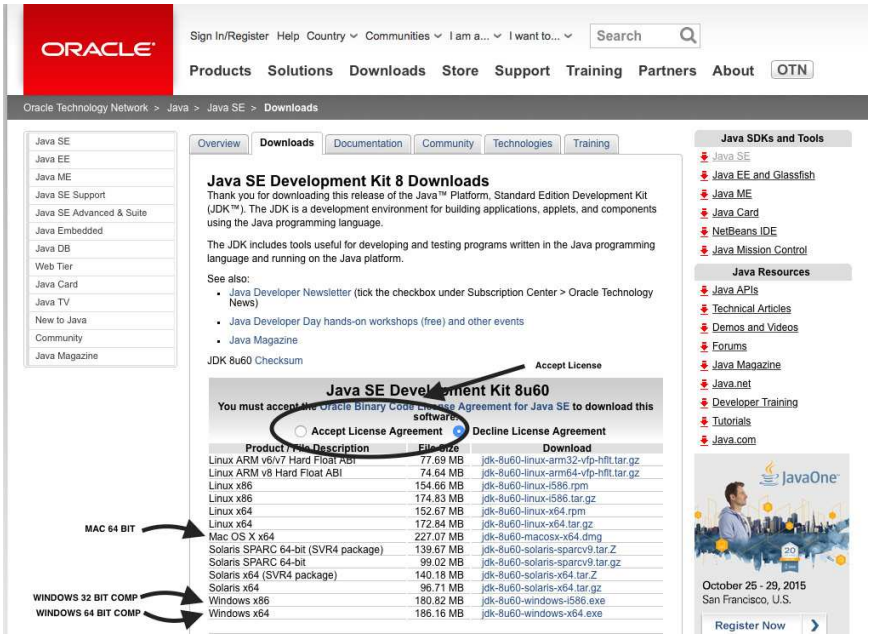


Figure A.1: Screenshot of Oracle's website

A.3 Self-Install for Mac

Step 1: Install Java

Ensure you are running Mac OSX 10.8 or later for Java to install properly, then go to Oracle's website:
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.
Under `Java SE Development Kit 8u60`, find the product/file description for Mac OS X x64. Before the website will allow you to download the DMG file, you will need to click "accept licensing agreement" right above the list of files. See figure A.1 for the location of the correct file and licensing agreement on the website.

To install Java, you will need to be an Admin on your computer or have the Admin password. Once the file, roughly 227 MB in size, finishes downloading, go to the Downloads folder and double click on the file (which should be named `jdk-8u60-macosx-x64.dmg`) to open the DMG file. A finder window should appear containing an icon of an open box and the name of the file. Double click on the box to launch the installation process. Follow the

instructions as presented on screen to complete the installation process.

Once completed, open a Terminal window. A Terminal can be found

- by using Spotlight and searching for "terminal"
- by clicking on your desktop so that you activate Finder. Then from the "Go" menu select Applications → Utilities → Terminal.

In the Terminal window, type `java -version` after the `$` and hit the Enter key. You which see `java version "1.8.0_60"` if Java is installed properly. If you have any issues installing Java, take a look at the step-by-step guide from Oracle:
https://docs.oracle.com/javase/8/docs/technotes/guides/install/mac_jdk.html

Step 2: Install a Text Editor

Step Two: Install a Text Editor We do not require you to use any particular text editor. There are many options available, but many students have found GEdit and Sublime2 as the best options.

Gedit is installed on all lab computers and will be used in lab sessions. For Mac we will be installing GEdit 3.2.6 which can be downloaded from this website:
http://ftp.gnome.org/pub/GNOME/binaries/mac/gedit/3.2/.
Download the appropriate version for your computer. Once downloaded, go to the Downloads folder and double click on the DMG file (named `gedit-3.2.6-3.dmg`) to open the installation screen. You will then drag GEdit's icon into the applications folder. Installation will not be complete until you open the application for the first time as an administrator.

Another popular text editor is Sublime2, which offers a free, non-time limited evaluation license. This is a more powerful and complex text editor then GEdit and may require more work to learn how to use its many features. You can download and install Sublime from this link:
http://www.sublimetext.com/2.

If you have an issue opening either of these applications, go to System Preferences → Security and Privacy and allow applications to run from unsigned developers for the duration of the installation. Then return the settings to their original values.

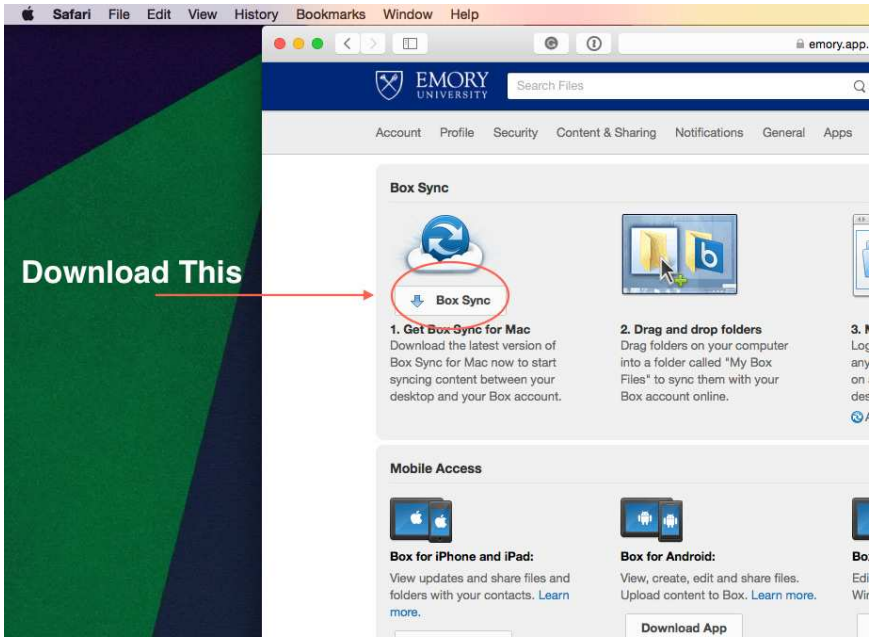


Figure A.2: Download Box Sync

Step 3: Setup a Project Folder through Box for Backups

Emory University offers each student 25GB of secure, cloud storage for free through Box.net. You can access this at:
https://emory.app.box.com/login.
You will login with your Emory UserID and password.

To ensure you do not lose any of your files, we **highly** recommend that you set up Box to sync with your local files, ensuring you have a backup just in case anything were to happen to your computer. This setup would mean that any changes made to the files on your computer would be saved to the cloud and could be accessed anywhere, including the Math/CS computers should you need to work on the lab computers in an emergency.

Once you are logged into Box, click on your name in the upper right hand corner to reveal the drop down menu. Click on the option "Get Box Sync" and follow the instructions on that page to finish the setup (see Figure A.2).

Once installed, Box will have created a "Box Sync" folder in your Documents folder. Inside this folder, create a new folder where you will keep your class documents named "cs170". Please note, anything you place in the "Box

Sync" folder will be saved to your Emory Box.net account while connected to the internet.

A.4 Self-Install for Windows

Step 1: Install Java

Ensure you are running Windows XP or later for Java to install properly, then go to Oracle's website:
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-213315
Under `Java SE Development Kit 8u60`, find the product/file description for your particular Windows computer. 32-bit computers should download x86 while 64-bit computers should download x64. If you are unsure what type of computer you have, you can follow this guide to figure it out:
http://windows.microsoft.com/en-us/windows7/find-out-32-or-64-bit.

Before the website will allow you to download the EXE file, you will need to click "accept licensing agreement" right above the list of files. See figure A.1 for the location of the correct file and licensing agreement on the website.

Once the file finishes downloading, find it in whatever folder your browser saves to. The file should be named `jdk-8.6.2-windows-x64.exe` for 64-bit computers and `jdk-8.6.2-windows-i586-i.exe` for 32-bit computers. Double click on the file to launch Java's installer. You will need to be the computer's Admin or have the Admin password to complete the installation.

Open the Command Prompt, which can be found by searching for "cmd" in the Start Menu in Windows 7 (Figure A.3, lower left) and 10 (Figure A.3, right) or on the Start Screen in Windows 8 (Figure A.3, upper left).

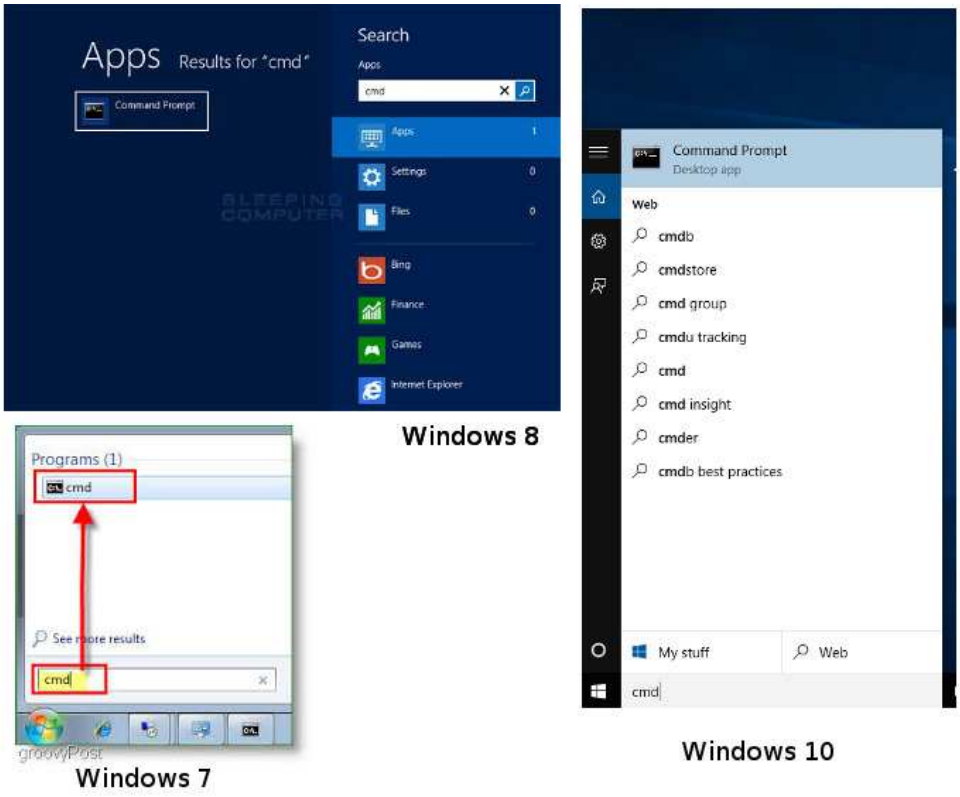


Figure A.3: Find the Command Prompt on Windows 7, 8, and 10

Figure A.4 shows the Windows Command Prompt, which is similar to a terminal.

If you have trouble finding Command Prompt read this guide:
http://winaero.com/blog/how-to-open-elevated-command-prompt-in-windows-10

In the Command window, type in the command `java -version`. You should see `java version "1.8.0_60"` if Java is installed properly. If you have any issues with the installation or issues with the PATH variable, read more about installing Java here:
http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-window

If you get an error when you attempt to run `java -version` instead of the version number, please follow this guide:

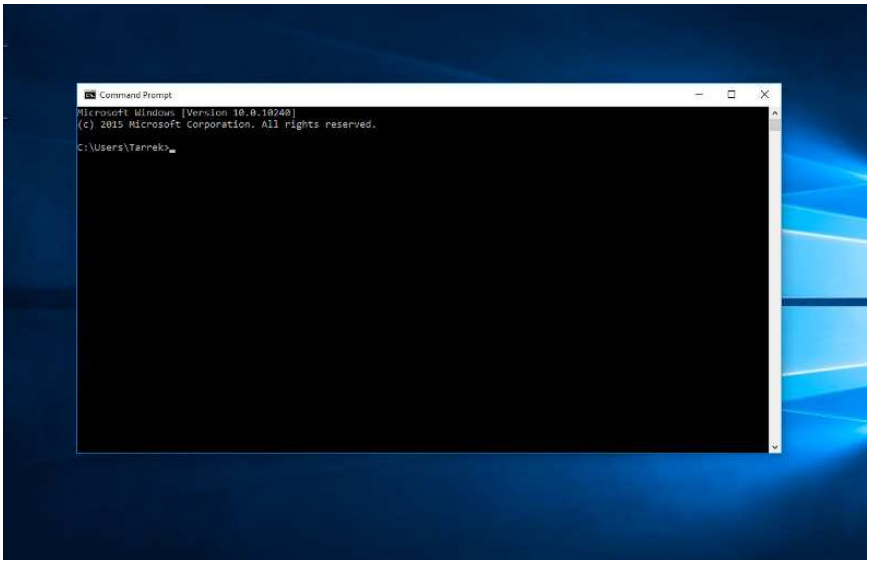


Figure A.4: Windows Command Prompt

http://www.abodeqa.com/2012/08/11/how-to-set-path/
to modify your classpath! Direct any additional issues and concerns to the class Piazza.

Step 2: Install a Text Editor

We do not require you to use any particular text editor. There are many options available, but many students have found GEdit and Sublime2 as the best options.

Gedit is installed on all lab computers and will be used in lab sessions. For Windows we will be installing GEdit 2.3 which can be downloaded from this website:
http://ftp.gnome.org/pub/GNOME/binaries/win32/gedit/2.30/.

Download the appropriate version for your computer. Once downloaded, go to the downloads folder and double click on the EXE file (named `gedit-setup-2.3.1-3.exe` to open the installation prompt. Follow the prompts to complete the installation process.

Another popular text editor is Sublime 2, which offers a free, non-time limited evaluation license. This is a more powerful and complex text editor then GEdit and may require more work to learn how to use its many features. You can download and install Sublime from this link:
http://www.sublimetext.com/2

Again, to install either of these text editor, you will need to have administrative privileges on your computer or know the admin password.

Step 3: Setup a Project Folder through Box for Backups

Emory University offers each student 25GB of secure, cloud storage for free through Box.net. You can access this at:
https://emory.app.box.com/login.
You will login with your Emory UserID and password.

To ensure you do not lose any of your files, we **highly** recommend that you set up Box to sync with your local files, ensuring you have a backup just in case anything were to happen to your computer. This setup would mean that any changes made to the files on your computer would be saved to the cloud and could be accessed anywhere, including the Math/CS computers should you need to work on the lab computers in an emergency.

Once you are logged into Box, click on your name in the upper right hand corner to reveal the drop down menu. Click on the option "Get Box Sync" and follow the instructions on that page to finish the setup (see Figure A.2).

Once installed, Box will have created a "Box Sync" folder in your Documents folder. Inside this folder, create a new folder where you will keep your class documents named "cs170". Please note, anything you place in the "Box Sync" folder will be saved to your Emory Box.net account while connected to the internet.

A.5 Remote Log-In

Professor Cheung has written an excellent guide for working remotely. Find it here:
http://www.mathcs.emory.edu/~cheung/Courses/RemoteAccess/index.html
This guide tells you how to setup both Mac and Windows computers. Remember that you need a constant internet connection to work in this manner.

Appendix B

Input and Output in Java

B.1 System objects

The `System` class provides methods and objects that get input from the keyboard, print text on the screen, and do file input and output (I/O). `System.out` is the object that displays on the screen. When you invoke `print` and `println`, you invoke them on `System.out`.

You can even use `System.out` to print `System.out`:

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

The result is:

```
java.io.PrintStream@80cc0e5
```

When Java prints an object, it prints the type of the object (`PrintStream`), the package where the type is defined (`java.io`), and a unique identifier for the object. On my machine the identifier is `80cc0e5`, but if you run the same code you will probably get something different.

There is also an object named `System.in` that makes it possible to get input from the keyboard.

B.2 Keyboard input

When a Java program starts running, the Java runtime system will initialize many variables in support for the running program.

One of these variables is the Java system variable: `System.in` which represents the keyboard input

The variable `System.in` is included in every Java program (you don't need to define or declare it).

The Java system variable `System.in` represents the keyboard and can capture what a user types on the command line in a Terminal window. This allows us to make our programs more flexible and interactive.

However, `System.in` is not in a format which is easy to capture. Moreover, it's not easy to discern what data the user is typing. If the user enters `123` do they intend for that to be a `String` or an `int`?

Java provides a class named `Scanner` to help with this. This class is a collection of methods which help us read and interpret data from a variety of sources, including files and user input via `System.in`.

In order to use a `Scanner` object in your program, you need to do three things:

1. Import the `Scanner` class definition.
2. Declare a new `Scanner` variable and initialize it to read from the keyboard (`System.in`).
3. Use the `Scanner` variable to read in a value that the user types.

B.2.1 Import Scanner class

By default, Java does not include the `Scanner` class for your use. If Java included all available classes, programs would grow to be unnecessarily large. So Java includes certain fundamental classes and allows a programmer to **import** other classes as needed. To import a class, we include the statement:

```
1    import java.util.Scanner;
```

outside of our class definition. Generally, `import` statements are the first line in your file.

This allows Java to load the Scanner class and have it ready for the programmer's use.

B.2.2 Make a Scanner variable

Next, we must make a `Scanner` typed variable and set it up to read from `System.in`.

```
    Scanner in = new Scanner(System.in);
```

The `Scanner` variable can be named anything, but a name like `in` or `input` `keyboard` helps us remember the purpose of this `Scanner` object.

B.2.3 Read user input

Once we have a `Scanner` variable, we can read the data that the user has typed. The only part that remains is what type of data the user is entering. We can use method calls in the `Scanner` class to read data from the keyboard input.

You can take a look at the `Scanner` class documentation here:
http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html
If you look in the "Method Summary" section, you will see many methods which begin with the word "next" like `nextBoolean()`, `nextDouble()`, and `nextInt()`. Each of these methods will read in some user input from the keyboard and try to format it as a specific Java datatype. We can then store that data into a correctly typed variable. For example:

```
    System.out.print("Please enter an integer: ");
    int x = in.nextInt();
```

After this code executes, the variable `x` will contain the integer the user typed.

This code also contains a prompt: "Please enter an integer:". Whenever you solicit data from a user, it's a good idea to tell them what you expect them to enter. If you omit the prompt, your program will not give any output and will not end. In reality, it will be waiting for the user to type something, but the user has no way of knowing that since you never told them!

But what happens if the user doesn't enter a valid integer at the prompt? What happens if the user types `abc123` instead? In this case, your program will have a runtime error: `InputMismatchException`.

Here's a complete program which reads a number from the user and then tells them what they entered.

```
import java.util.Scanner;

public class UserInput {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

```

```
7       System.out.print("Please enter an integer: ");
8       int x = in.nextInt();
9       System.out.println("You entered " + x);
10    }
11  }
```

Program B.1: UserInput.java

Play around with this program. What happens if you enter a value like `4.5` or `"123"`?

The benefits of reading user input are obvious. We no longer have to compile every time we want to test our code with a new value! We just re-run our program and enter a new value at the prompt.

Appendix C

Program development

C.1 Strategies

I present different program development strategies throughout the book, so I wanted to pull them together here. The foundation of all strategies is **incremental development**, which goes like this:

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 tests 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. And 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 wrong, you are confronted with the conflict (and have a chance to correct it) before you write a lot of bad code.

The challenge of incremental development is that is it not easy to figure out a path from the starting place to a complete and correct program. To help with that, there are several strategies to choose from:

Encapsulation and generalization: If you don't know yet how to divide the computation into methods, start writing code in `main`, then look for coherent chunks to encapsulate in a method, and generalize them appropriately.

Rapid prototyping: If you know what method to write, but not how to write it, start with a rough draft that handles the simplest case, then test it with other cases, extending and correcting as you go.

Bottom-up: Start by writing simple methods, then assemble them into a solution.

Top-down: Use pseudocode to design the structure of the computation and identify the methods you'll need. Then write the methods and replace the pseudocode with real code.

Along the way, you might need some scaffolding. For example, each class should have a `toString` method that lets you print the state of an object in human-readable form. This method is useful for debugging, but usually not part of a finished program.

C.2 Failure modes

If you are spending a lot of time debugging, it is probably because you are using an ineffective development strategy. Here are the failure modes I see most often (and occasionally fall into):

Non-incremental developement: If you write more than a few lines of code without compiling and testing, you are asking for trouble. One time when I asked a student how the homework was coming along, he said, "Great! I have it all written. Now I just have to debug it."

Attachment to bad code: If you write more than a few lines of code without compiling and testing, you may not be able to debug it. Ever. Sometimes the only strategy is (gasp!) to delete the bad code and start over (using an incremental strategy). But beginners are often emotionally attached to their code, even if it doesn't work. The only way out of this trap is to be ruthless.

Random-walk programming: I sometimes work with students who seem to be programming at random. They make a change, run the program, get an error, make a change, run the program, etc. The problem is that there is no apparent connection between the outcome of the program and the change. If you get an error message, take the time to read it. More generally, take time to think.

Compiler submission: Error messages are useful, but they are not always right. For example, if the message says, "Semi-colon expected on line 13," that means there is a syntax error near line 13. But putting a semi-colon on line 13 is not always the solution. Don't submit to the will of the compiler.

The next chapter makes more suggestions for effective debugging.

Appendix D

Debugging

The best debugging strategy depends on what kind of error you have:

- Syntax errors are produced by the compiler and indicate that there is something wrong with the syntax of the program. Example: omitting the semi-colon at the end of a statement.
- Exceptions are produced if something goes wrong while the program is running. Example: an infinite recursion eventually causes a `StackOverflowException`.
- Logic errors cause the program to do the wrong thing. Example: an expression may not be evaluated in the order you expect, yielding an unexpected result.

The following sections are organized by error type; some techniques are useful for more than one type.

D.1 Syntax errors

The best kind of debugging is the kind you don't have to do because you avoid making errors in the first place. In the previous section, I suggested development strategies that minimize errors and makes it easy to find them when you do. The key is to start with a working program and add small amounts of code at a time. When there is an error, you will have a pretty good idea where it is.

Nevertheless, you might find yourself in one of the following situations. For each situation, I make some suggestions about how to proceed.

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 often gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it reports spurious errors.

Only the first error message is truly 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.

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 carefully hidden kernel of information.

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 don't find the error quickly, take a breath and look more broadly at the entire program. Make sure the program is indented properly; that makes it easier to spot syntax errors.

Now, start looking for common 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 curly-braces).

4. Make sure that any strings in the code have matching quotation marks. Make sure that you use double-quotes for Strings and single quotes for characters.

5. For each assignment statement, make sure that the type on the left is the same as the type on the right. Make sure that the expression on the left is a variable name or something else that you can assign a value to (like an element of an array).

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 value 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 get a 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 development environment.

If you have examined the code thoroughly, and you're sure the compiler is compiling the right code, it is time for desperate measures: **debugging by bisection**.

- Make a copy of the file you are working on. If you are working on `Bob.java`, make a copy called `Bob.java.old`.
- Delete about half the code from `Bob.java`. Try compiling again.
  - If the program compiles now, 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 ugly, but it goes faster than you might think, and it is very reliable.

I did what the compiler told me to do, but it still doesn't work.

Some compiler messages come with tidbits of advice, like "class Golfer must be declared abstract. It does not define int compareTo(java.lang.Object) from interface java.lang.Comparable." It sounds like the compiler is telling you to declare Golfer as an abstract class, and if you are reading this book, you probably don't know what that is or how to do it.

Fortunately, the compiler is wrong. The solution in this case is to make sure `Golfer` has a method called `compareTo` that takes an `Object` as a parameter.

Don't let the compiler lead you by the nose. Error messages give you evidence that something is wrong, but the remedies they suggest are unreliable.

D.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 suggestions helps, you might not 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 you know which loop it is, 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 see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loop keeps going, you will see the values of `x` and `y` and you might figure out why they are not updated correctly.

Infinite recursion

Most of the time an infinite recursion will cause the program to throw a `StackOverflowException`. But if the program is slow it may take a long time to fill the stack.

If you know which method is causing an infinite recursion, check that there is a base case. There should be some condition that makes the method return without making a recursive invocation. If not, 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 see a few lines of output every time the method is invoked, and you see the values of the parameters. If the parameters are not moving toward the base case, you might see 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 prints a trace of each method as it is invoked.

You can also print the arguments each method receives. When you run the program, check whether the values are reasonable, and check for one of the most common errors—providing arguments in the wrong order.

When I run the program I get an Exception.

When an exception occurs, Java 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 was running, the method that invoked it, the method that invoked *that*, and so on.

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 which 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."

FileNotFoundException: This means Java didn't find the file it was looking for. If you are using a project-based development environment like Eclipse, you might have to import the file into the project. Otherwise make sure the file exists and that the path is correct. This problem depends on your file system, so it can be hard to track down.

ArithmeticException: Occurs when something goes wrong during an arithmetic operation, most often division by zero.

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. As you develop a program, you should write code to generate concise, informative visualizations of what the program is doing.

To simplify the program, 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.

Also, clean up the code. Remove dead code and reorganize the program to make it easier to read. For example, if you suspect that the error is in a deeply-nested part of the program, rewrite that part with simpler structure. If you suspect a large method, split it into smaller methods and test them separately.

The process of finding the minimal test case often 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.

Reorganizing the program 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.

D.3 Logic errors

My program doesn't work.

Logic errors are hard to find because the compiler and the run-time system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the code and the behavior you get. You need a hypothesis about what the program is actually doing. 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. See "Flow of execution" above.

- 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 unexpected effect? Make sure you understand the code, especially if it invokes Java methods. Read the documentation for those methods, and try them out with simple test cases. They might not do what you think they do.

To program, you need a mental model what your code does. If it doesn't do what you expect, the problem might not be the program; it might be in your head.

The best way to correct your mental model is to break the program into components (usually the classes and methods) and test them independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Here are some common logic errors to check for:

- Remember that integer division always rounds down. If you want fractions, use `doubles`.

- Floating-point numbers are only approximate, so don't rely on perfect accuracy.

- More generally, use integers for countable things and floating-point numbers for measurable things.

- 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 syntactically legal and semantically wrong.

- When you apply the equality operator (`==`) to an object, it checks identity. If you meant to check equivalence, you should use the `equals` method.

- For user defined types, `equals` checks identity. If you want a different notion of equivalence, you have to override it.

- Inheritance can lead to subtle logic errors, because you can run inherited code without realizing it. See "Flow of Execution" above.

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:

```
rect.setLocation(rect.getLocation().translate(
                -rect.getWidth(), -rect.getHeight()));
```

Can be rewritten as

```
int dx = -rect.getWidth();
int dy = -rect.getHeight();
Point location = rect.getLocation();
Point newLocation = location.translate(dx, dy);
rect.setLocation(newLocation);
```

The explicit version is easier to read, because the variable names provide additional documentation, and easier to debug, because you can check the types of the temporary 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, to evaluate $\frac{x}{2\pi}$, you might write

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and they are evaluated from left to right. This expression computes $x\pi/2$.

If you are not sure of the order of operations, use parentheses to make it explicit.

```
double y = x / (2 * Math.PI);
```

This version is correct, and more readable for other people who haven't memorized the order of operations.

My method 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 value before returning. Again, you can use a temporary variable. For example, instead of

```
public Rectangle intersection(Rectangle a, Rectangle b) {
    return new Rectangle(
        Math.min(a.x, b.x),
        Math.min(a.y, b.y),
        Math.max(a.x+a.width, b.x+b.width)-Math.min(a.x, b.x)
        Math.max(a.y+a.height, b.y+b.height)-Math.min(a.y, b.y) );
}
```

You could write

```
public Rectangle intersection(Rectangle a, Rectangle b) {
    int x1 = Math.min(a.x, b.x);
    int y2 = Math.min(a.y, b.y);
    int x2 = Math.max(a.x+a.width, b.x+b.width);
    int y2 = Math.max(a.y+a.height, b.y+b.height);
    Rectangle rect = new Rectangle(x1, y1, x2-x1, y2-y1);
    return rect;
}
```

Now you have the opportunity to display any of the intermediate variables before returning. And by reusing `x1` and `y1`, you made the code smaller, too.

My print statement isn't doing anything

If you use the `println` method, the output is displayed immediately, but if you use `print` (at least in some environments) the output gets stored without being displayed until the next newline. If the program terminates without printing a newline, you may never see the stored output.

If you suspect that this is happening to, change some or all of the `print` statements to `println`.

I'm really, really stuck and I need help

First, get away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backwards").
- Sour grapes ("this program is lame anyway").

If you suffer 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 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 let my mind wander. Good places to find bugs are trains, showers, and bed.

No, I really need help.

It happens. Even the best programmers get stuck. Sometimes you need a fresh pair of eyes.

Before you bring someone else in, make sure you have tried the techniques described above. Your 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, give them the information they need.

- What kind of bug is it? Syntax, run-time, or logic?
- 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?
- 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 have you tried, and what have you learned?

By the time you explain the problem to someone, you might see the answer. This phenomenon is so common that some people recommend a debugging technique called "rubber ducking." Here's how it works:

1. Buy a standard-issue rubber duck.
2. When you are really stuck on a problem, put the rubber duck on the desk in front of you and say, "Rubber duck, I am stuck on a problem. Here's what's happening..."

3. Explain the problem to the rubber duck.
4. See the solution.
5. Thank the rubber duck.

I am not kidding. See http://en.wikipedia.org/wiki/Rubber_duck_debugging.

I found the bug!

When you find the bug, it is usually obvious how to fix it. But not always. Sometimes what seems to be a bug is really an indication that you don't understand the program, or there is an error in your algorithm. In these cases, you might have to rethink the algorithm, or adjust your mental model. Take some time away from the computer to think, work through test cases by hand, or draw diagrams to represent the computation.

After you fix the bug, don't just start in making new errors. Take a minute to think about what kind of bug it was, why you made the error, how the error manifested itself, and 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.

Appendix E

Searching and Sorting

E.1 Overview

Searching through data is something that all scientists do frequently. We want to know things like whether or not a particular value occurs in a set of data or how many times a certain value occurs. This task is easy when you only have a small set of data to evaluate. But with modern technology and digital storage, we can now store huge amounts of data. It's much more difficult to look through terabytes of data in search of a given value.

So computer scientists have developed many algorithms to search through data. Different algorithms perform differently and are better suited for different tasks. For example, one algorithm may perform well on data which is already almost in order while another algorithm my perform well on data which only contains a few unique values and contains mostly duplicate data.

However, one search algorithm that is quite useful and efficient is called a `binary search`. This algorithm only works if the data is sorted in some order (usually smallest value to largest value). If the data isn't or can't be sorted, we have to revert to a `linear search` which isn't nearly as efficient. Therefore, computer scientists spend lots of time talking about `sorting` data as well. Once data is in sorted order, we have some very efficient searches we can do. Sorting means that we take a collection of unordered data and we put it in order.

In this section, we'll talk about the linear and binary search algorithm first, and then we'll discuss a few sorting algorithms. However, we won't spend much time on the code to implement these algorithms. The code for

any of these algorithms is easy to find using your choice of search engine. Here, we're interested in understanding how the algorithms work on a step-by-step basis.

We'll discuss three different sorts: Insertion Sort, Selection Sort, and Bubble Sort. These sorts are easy to understand and require no coding skills beyond basic things like methods, conditional logic, and loops. There are lots of other sorting algorithms too. If you continue in your study of computer science, you'll discuss many sorting algorithms like Merge Sort, Quick Sort, and others.

E.2 Linear and Binary Search

The algorithm for a binary search is relatively easy and intuitive to understand. In fact, you instinctively revert to it when you play a simple guessing game. Consider a game where Player 1 thinks of a number between 1-100 and Player 2 tries to guess the number. Let's consider two scenarios:

- When Player 2 guesses, Player 1 just answers "Yes" or "No" to indicate whether or not Player 2 guessed correctly.
- When Player 2 guesses, Player 1 just answers "Yes" or "No" to indicate whether or not Player 2 guessed correctly AND Player 1 tells Player 2 if the number is higher or lower then Player 2's guess.

In the first scenario, a conversation might go like this:
Player 1: I'm thinking of a number between 1 and 100.
Player 2: Is it 1?
Player 1: No.
Player 2: Is it 2?
Player 1: No.
...
Player 2: Is it 100?
Player 1: Yes! You got it.

Since Player 1 provides no additional information, there is no way to speed up the search for the chosen number. In a worst case scenario, Player 2 might have to guess 100 numbers before hitting on Player 1's number. However, consider the second scenario above when we might have a conversation like

this:
Player 1: I'm thinking of a number between 1 and 100.
Player 2: Is it 50?
Player 1: No, higher.
Player 2: Is it 75?
Player 1: No, lower.
And so forth.

After being told the number was more than 50, would Player 2 ever guess a number like 20 or 30? Of course not. With a single guess of 50, Player 2 eliminates half of the possible 100 numbers that Player 1 might have been thinking of. There are only 50 possible numbers left (values 51-100), and with a second guess of 75, we eliminate half of those! The only possible candidates are the values 76-100! We narrow down the field of 100 possibilities quite rapidly using this strategy.

The first strategy is like a linear search. We must use a linear search when the data in an array is unsorted, like in Figure E.1 below. If we want to know if the value 22 is in the array, we have no choice but to examine all the elements in the array until we either find the element (at which point we can stop searching) or until we reach the end of the array and can definitively say that the value does not exist in the array.

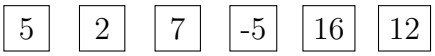


Figure E.1: Unsorted Array Data

However, once the data is sorted, we can employ a more efficient strategy: eliminating half of the array at each "guess". To implement this strategy, we need to determine what to guess "in the middle" of our data. When Player 2 guessed "50" in our guessing game, it was because she had 100 numbers, and she knew that 50 was the middle value. When working with arrays, we don't know what the middle value will be, but we do know how many elements are in the array. Figure E.2 shows an array sorted data which has 9 elements. We can easily calculate where the middle of the array is with the expression `a.length/2` which will give us 4, and indeed, `a[4]` is in the exact middle of our array. This is a good place to start.

Figure E.2: Sorted Array Data

After that, we compare the value we're searching for (let's say 15) to the value in the middle of our array: `a[4]` or 5. Since we know 15 is greater than 5 and we know the array is sorted, there is no need for us to examine the elements at `a[0]-a[3]` because we know they will be less than 5! We then repeat the strategy by splitting the remaining array (`a[6]-a[9]`) in half again and examining the element in the middle. In this case, there are 4 elements so there isn't a way to evenly partition things. The binary search algorithm can use either `a[7]` (the value 12) or `a[8]` (the value 16) as the "middle" element.

We compare our search value (15) to `a[8]` and discover that it is less than 16, so if 15 occurs, it must be between `a[5]` and `a[8]`. We can repeat this process of splitting the array in half and examining the middle element until there are no further elements left in the array and we can definitively say that 15 does not appear in the array.

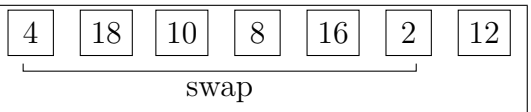
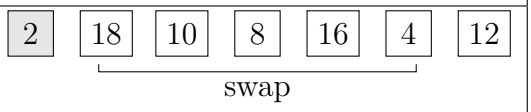
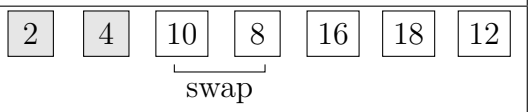
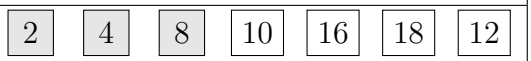
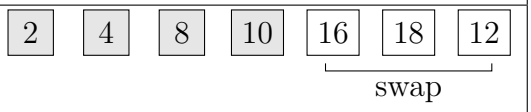
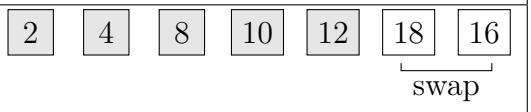
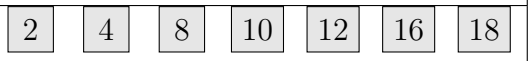
You can see why binary search is preferred. It uses many fewer comparisons than a linear search. However, it requires time and (computational) power to turn an unsorted array into a sorted array, so there is often a trade off. If we only want to search a set of data for a single value, it may make sense to just do a linear search. However, if we want to search for many values in the data set, it will probably be worth the time to sort the data and then use the binary search algorithm repeatedly.

Next, we'll examine three different sorting algorithms which can change an unsorted array into a sorted one.

E.3 Selection Sort

The selection sort algorithm is easy to understand which is why we choose to discuss it in this course. It is often a solution which students intuitively articulate when asked to describe how to sort an array of numbers.

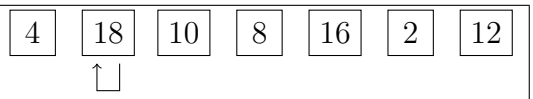
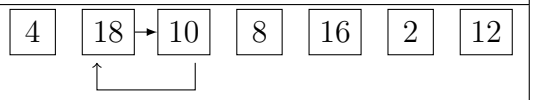
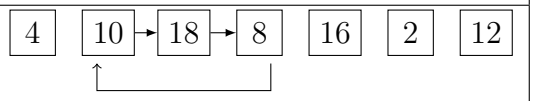
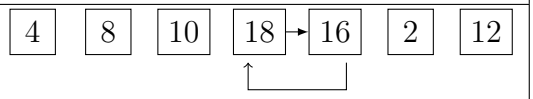
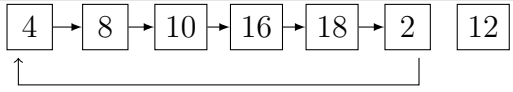
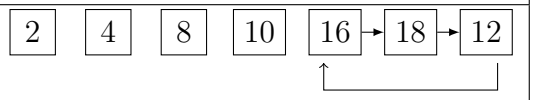
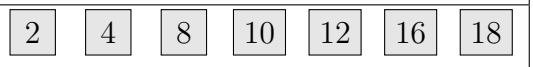
Selection sort works by searching an array for the smallest value and then swapping that value to the front of the array. We progress through the unsorted portion of the array until all the elements are moved into sorted order. The values shaded gray have been sorted into position.

Select 2 (smallest value) and swap it with 4 (first value) in the list	
Now 2 is sorted. Select 4 (smallest value in remaining values) and swap it with 18 (first unsorted value in remaining values)	
Now 2 and 4 are sorted. Select 8 (smallest value in remaining values) and swap it with 10 (first unsorted value in remaining values)	
Now 2, 4, and 8 are sorted. Select 10 (smallest value in remaining values) and swap it with 10 (first unsorted value in remaining values). This means the list remains unchanged.	
Now 2, 4, 8, and 10 are sorted. Select 12 (smallest value in remaining values) and swap it with 16 (first unsorted value in remaining values).	
Now 2, 4, 8, 10, and 12 are sorted. Select 16 (smallest value in remaining values) and swap it with 18 (only remaining unsorted value).	
Since there is only 1 value remaining, it is also sorted, and we are finished.	

E.4 Insertion Sort

Unlike Selection Sort, Insertion Search does not involve searching through the array for the smallest value. Instead, it takes the elements in order and inserts them into their place in a sorted list. Other elements may have to be shifted to make room for a number which needs to be inserted.

The following table shows the Insertion Sort algorithm, step-by-step. Unlike the previous example, we don't really know that the elements are in their final, sorted order until the last step. After all, another smaller element might be found later which would necessitate shifting the other elements to make room for it. In the following example, note that it isn't until the last step that we know our list of numbers is sorted and the list is shaded gray.

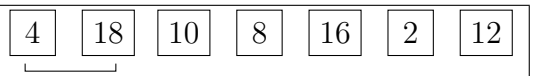
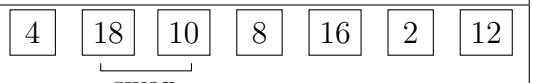
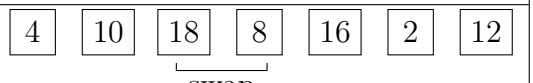
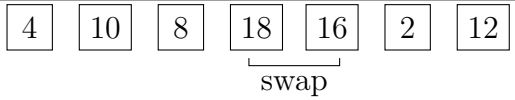
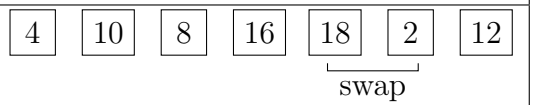
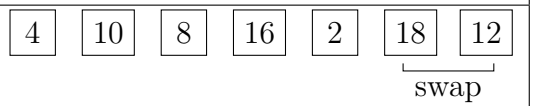
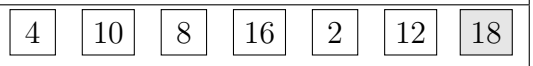
Initially, the first element (4) is considered to be sorted. We insert the next element (18) in relation to the value 4. Since 18 is greater than 4, no movement is required	
The sorted list is 4 and 18. Insert 10 into this list. We must shift 18 to the right to make room for 10.	
The sorted list is 4, 10, 18. Insert 8 into this list, and shift 18 and 10 down.	
The sorted list is 4, 8, 10, 18. Insert 16 into this list, and shift 18 down.	
The sorted list is 4, 8, 10, 16, 18. Insert 2 into this list, and shift all other numbers down.	
The sorted list is 2, 4, 8, 10, 16, 18. Insert 12 into this list, and shift 16 and 18 down.	
Since we have processed all the elements in our array, we know that it is now sorted.	

E.5 Bubble Sort

Unlike the previous two sorts, bubble sort relies on what we call "pairwise comparisons." This means that we need to compare pairs of numbers which occur next to each other in our list. We swap the two values if we find they are out of order.

Because we only compare 2 values to each other, this sort requires more iterations through our list of numbers. Both Insertion Sort and Selection Sort operated by comparing a value to multiple other values before placing it back into the list.

The next page shows how 1 pass of this algorithm would work:

First, we compare the first and second values (4 and 18). We find they are in order, so we don't need to swap them.	
Next, we compare the second and third values (18 and 10). We find they are not in order, so we swap them.	
Compare the third and fourth values (18 and 8). We find they are not in order, so we swap them.	
Compare the fourth and fifth values (18 and 16). We find they are not in order, so we swap them.	
Compare the fifth and sixth values (18 and 2). We find they are not in order, so we swap them.	
Compare the sixth and seventh values (18 and 12). We find they are not in order, so we swap them.	
At this point, we know that exactly 1 value (18) is in place. We need to continually repeat this process until all values are in sorted order.	

Bubble sort is so named because the largest elements "bubble" to the top, much as a bubble floats to the top of the water. Each pass over the list sorts another element into place. The element which is sorted will always be the largest value in the remaining unsorted elements.

You can reverse this algorithm to place the smallest number by starting at the end of the list and doing pairwise comparisons and swaps while moving towards the beginning of the list. Regardless of whether you code this algorithm to sort by the largest or smallest value, it requires multiple passes over the list of values to ensure that all the values are sorted.

Index